Configurable Simultaneously Single-Threaded (Multi-)Engine Processor

by

Anita Tino
Bachelor of Engineering (B.Eng), Ryerson University, 2009
Master of Applied Science (M.A.Sc), Ryerson University, 2011

A dissertation presented to Ryerson University
in partial fulfilment of the
requirements for the degree of

Doctor of Philosophy

in the program of
Electrical and Computer Engineering

Toronto, Ontario, Canada, 2017
©Anita Tino, 2017

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A DISSERTATION

I hereby declare that I am the sole author of this thesis dissertation. This is a true copy of the dissertation, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this dissertation to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this dissertation by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my dissertation may be made electronically available to the public.

-Anita Tino

Configurable Simultaneously Single-Threaded (Multi-)Engine Processor

Anita Tino
Doctor of Philosophy, 2017,
Electrical and Computer Engineering,
Ryerson University

Abstract

As the multi-core computing era continues to progress, the need to increase single-thread performance, throughput, and seemingly adapt to thread-level parallelism (TLP) remain important issues. Though the number of cores on each processor continues to increase, expected performance gains have lagged. Accordingly, computing systems often include Simultaneously Multi-Threaded (SMT) processors as a compromise between sequential and parallel performance on a single core. These processors effectively improve the throughput and utilization of a core, however often at the expense of single-thread performance as threads per core scale. Accordingly, applications which require higher single-thread performance must often resort to single-thread core multi-processor systems which incur additional area overhead and power dissipation. In attempts to improve single- and multi-thread core efficiency, this work introduces the concept of a Configurable Simultaneously Single-Threaded (Multi-)Engine Processor (ConSSTEP). ConSSTEP is a nuanced approach to multi-threaded processors, achieving performance gains and energy efficiency by invoking low overhead reconfigurable properties with full software compatibility. Experimental results demonstrate that ConSSTEP is able to increase single-thread Instructions Per Cycle (IPC) up to 1.39x and 2.4x for 2-thread and 4-thread workloads, respectively, improving throughput and providing up to 2x energy efficiency when compared to a conventional SMT processor.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Kaamran Raahemifar, for his guidance and support of unconventional approaches and ideas throughout my PhD studies. I am truly grateful for the sincere trust and patience he has placed in me to develop the work presented in this thesis. I would like to thank the Ontario Graduate Scholarship (OGS) program and the Faculty of Engineering, Architecture, and Science (FEAS) for their financial support throughout my PhD studies. A special thanks to my thesis defense committee - Dr. Lev Kirischian, Dr. Ebrahim Bagheri, Dr. Issaac Woungang, and Dr. George Stamoulis, for their valuable feedback and comments which have greatly enhanced the quality of this thesis.

I would also like to thank Dr. Gul N. Khan for his advice and support throughout my graduate studies, and especially for granting me with opportunities that have truly enhanced my experience at Ryerson. To Ryerson's faculty and staff, I would like to thank them for sharing their wisdom and knowledge throughout the years, making my Ryerson experience memorable. Also a special thanks to Jason, Luis, Dan, and Nipin for all their guidance, assistance, and laughs throughout the years. Sincere thanks to Luis for his help on the development of this written thesis.

Finally to those closest to me: To my mom, sister Cassandra, my loving grandparents and family, I would like to sincerely thank them for their constant support and encouragement to pursue my studies, and especially for tolerating my never ending university workload. I am truly blessed to have such a supportive and loving family, and would not be where I am today without them. To my best friend Cristina who has been nothing but supportive, encouraging, and loving since day one and especially throughout my graduate studies - words cannot express how truly grateful I am. And finally to Matthew - we've developed an exponential closeness throughout the past years. You push me to be the best I can be with endless love and encouragement, and I am sincerely thankful to have you in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Processor Performance Plateau

Computers and similar devices play an essential role in our daily lives, whether it be our laptops, tablets, and/or cell phones etc. The past 10 to 15 years within computing have demonstrated major technological progression. One of the most influential roles in such devices is the role of the processor, i.e. the computer's "brain". A difficult challenge within computing is how to provide an increase in performance and energy efficiency per processor generation. Although performance gains have been prominent throughout the history of computing, we have observed a plateau in more recent multi-core computing generations. Accordingly, Fig. 1.1 displays eminent Intel processors from the dawn of the first on-chip cache processor, with the x axis representing the performance gain per processor generation with respect to the previous generation. The figure was originally captured by Borkar and Chien [1], and extended here for applicability of Central Processing Unit (CPU) trends to date. As verified in the figure, the past few processor generations have demonstrated negligible performance gains.

Fig. 1.1 also demonstrates industry's recent continuous emphasis on the multicore generation. An application however consists of both sequential and parallel sections. Therefore, although the parallelizable sections continue to be the main focus of recent computing trends, applications may only benefit from such multi-cores to a certain extent as applications are still bound by their sequential counterparts, supported by the theory of Amdahl's law[2]. Furthermore, only so much parallel

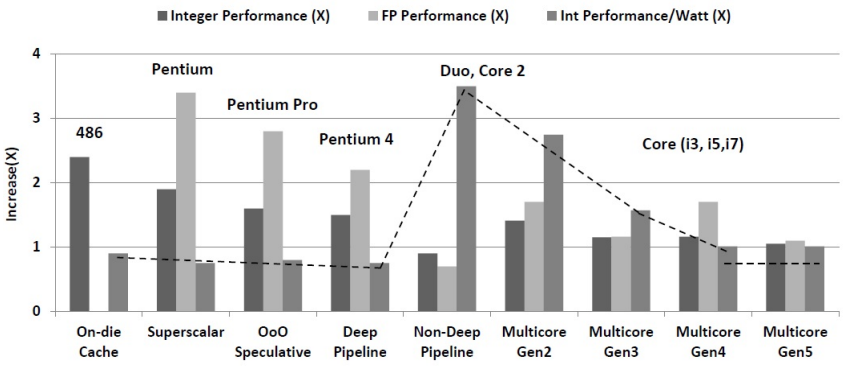


Figure 1.1: Architectural Advances and Energy Efficiency

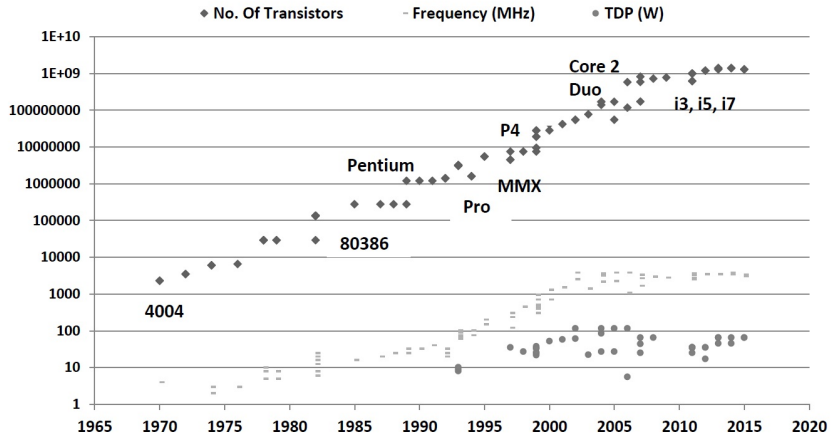


Figure 1.2: CPU Trends

performance may be gained from multi-cores as applications possess variable and/or limited thread-level parallelism (TLP) [3].

Majority of single-thread (sequential) performance gains have been achieved by a reliance on transistor scaling. Specifically, transistor scaling has accounted for almost three-orders of magnitude in performance improvement over the past 20 years of processor history (i.e. increasing a chip's attainable frequency to increase the instructions processed per second)[1]. Accompanying Fig. 1.1, Fig. 1.2 presents CPU trends from the 1970's until 2016, illustrating the transistor count, frequency of operation, and thermal design power of various microprocessors. As seen in the figure, we have now reached a plateau in transistor and frequency scaling, correlating to the performance plateau observed in Fig. 1.1. Consequently very little single-thread performance improvement has actually been attained due to architectural optimizations on chip, and accordingly such trends scale as more and more cores

are integrated onto a die (considering homogeneous multi-cores, which replicate the same core on a die). As industry can no longer rely on transistor scaling, the need to increase processor performance through architectural optimizations and/or alternate designs and technologies are therefore the only viable solutions.

In response to the multi-core processor plateau, computing has now steered towards the heterogeneous computing era which integrates cores of various types on a single chip. Such a heterogeneous approach encompasses both latency- and throughput-oriented cores on a die to provide efficient single-thread performance and thread-level parallelism, respectively. Although these systems cater to various workloads, a single core's architecture must still be considered within a multi-core system as it scales to provide performance gains. Accordingly, there is much ingenuity to be discovered if transistor budgets are allocated to improving and sophisticating processing element performance, versus adopting the mantra of integrating more cores and memory on a single die[4]. Such a concept provides motivation for this thesis work.

1.1.2 Computing Model Classifications

Computing systems may be divided into three general categories: Computing Systems with Programmable Procedure (CSPP), computing systems with Application Specific Processing (ASP), and Reconfigurable Computing Systems (RCS) [5]. Each type of computing system has their advantages and limitations depending on the workload considered. Accordingly, a computing architecture, A, may be defined as a triple set of [5]:

1. Functional components (C_i)

2. Communication links between the components ($L_{i,j}$), and

3. Functional procedures associated with C and L ($P_{i,j}$)

Depending on the computing system, any of the given elements in the triple set may have fixed, x, or programmable, $\sim$ x, functionality.

Conventional processors are considered as CSPP models, such that $A_{CSPP} = \{C_i, L_{i,j}, \sim P_{i,j}\}$ [5]. In this case, its links and components are fixed, however

the computing system executes *programmable* procedures (programs). The programmable procedures are more commonly referred to as *general purpose applications* which are designed to execute a given algorithm as coded by the programmer. The CSPP model provides a universal computing system with its application flexibility and low-cost manufacturing process to process data. Aside from its technological reliance mentioned previously (i.e. frequency, transistor scaling etc), limitations of the model include the processing overhead incurred per instruction. That is, a processor requires that a program to be divided into a sequence of elementary subtasks, referred to as **instructions** [5], formatted according to the specifications supported by the underlying architecture. An instruction consists of an operation, where instructions collectively execute the desired program. Specifically, each instruction specifies a basic operation, its data input operands, and an output operand. The exact formatting and data support for the instructions is dictated by the processor's target Instruction Set Architecture (ISA)[1].

The CSPP model requires several steps to process an instruction: initiation and decoding, input source data delivery, data processing, and result storage [5]. These steps are overlapped to create a *pipeline* which the processor uses to simultaneously process multiple successive instructions of a program. Such steps however create much control logic overhead in CSPP models as they must process such elementary data operations at the expense of time, energy, and hardware logic overhead. Therefore many clock cycles in a conventional processor pipeline[2] are spent on servicing and processing control information per instruction versus actually executing instruction data.

Heterogeneous systems integrate a variety of CSPP models and other compute models to achieve performance improvements. The other compute models integrated into such heterogeneous systems however are usually optimized to execute certain types of applications/algorithms and data structures. Such a compute model is referred to as an Application Specific Integrated Circuit (ASIC) or ASC, such that $A_{ASC} = \{C_i, L_{i,j}, P_{i,j}\}$. Accordingly, all components in the computing model are fixed and the system's efficiency is at the expense of no application flexibility. Performance gain and energy efficiency is therefore possible as the circuits were designed and optimized to execute a particular algorithm, versus CSPPs which were designed

[1] Chapter 1.1.3 provides further information on ISAs

[2] Chapter 2.2 provides more insight on the advantages and drawbacks of pipelining

to execute any application. In order to compromise between conventional CPUs and ASICs, Reconfigurable Computing Systems (RCS) may also be used to provide total architectural flexibility such that $A_{RCS} = \{\sim C_i, \sim L_{i,j}, \sim P_{i,j}\}$. RCS models may rapidly adapt to any application using its programmable links and components, however at the expense of hardware, power, and re-programmability overhead. Intel's recent acquisition of Altera however suggests that this RCS approach will soon be underway, integrating CSPP models with FPGA-based accelerators. Heterogeneous systems may also integrate Graphic Processing Units (GPUs) on a die to provide throughput-oriented execution. Such a core however may be classified as a CSPP, demonstrating the same general advantages and limitations of the model.

Each of the three computing models discussed in this section have their advantages and limitations. Considering an objective to improve the CSPP model, the non-flexible characteristics exhibited by the ASIC model would not benefit the programmable procedure objective sought by such general purpose processors. The RCS model however poses as an attractive solution to overcome the shortcomings of a processor's architecture. As mentioned previously, the advantages of reconfigurability are at the expense of the additional latencies incurred by flexible logic, hardware overhead (in comparison to a dedicated circuitry), and the additional memory required to store such configuration logic. These configuration latency penalties would therefore be detrimental to increasing the performance of general purpose processors. Such penalties however are primarily problematic due to the way the CSPP model was designed, and not inherently in computing itself. It may therefore be possible to revisit the design fundamentals of a conventional processor by using current computing knowledge to re-design the CSPP model, originally created over half a century ago under vastly different technological circumstances. Accordingly, a nuanced processor architecture with an adaptive computing model for programmable procedures may be possible.

1.1.3 Limitations of ISAs and Compilers

In addition to the conventional processor bottlenecks discussed, processor architectures are also stagnated by the concept of an Instruction Set Architecture (ISA). Conventional processors execute instructions according to a given ISA which provides a programmer with a simplified view of the microarchitecture – the underlying implementation and organizational details of the processor's hardware [5]. As pre-

viously mentioned, the ISA specifies the instruction formats, operands, operations, addressing, and data types that are supported by the processor.

A compiler's role is to translate a program into a set of instructions. Thus when the compiler accepts an application as input, it generally has an unrestricted view of the program with knowledge of instruction dependencies, operand usage, and lifetimes [6]. Such information however is eliminated during the last compilation stage which generates machine code for the processor according to the ISA. Consequently, the compiler is only able to express the program through basic instructions, and the processor hardware must dedicated a considerable portion of its logic to (dynamically) rediscover program characteristics which were known by the compiler. The hardware structures used to compensate for the ISA are at the expense of a core's power, area overhead, and latency per die. ISAs therefore reflect a division of labour between hardware and software that was created decades ago, again under vastly different technological circumstances [7].

It is possible however for processors to employ a smarter compilation process by resolving dependencies statically to eliminate complex hardware logic, while supporting wide-issue execution (i.e. the execution of many simultaneous instructions) per cycle. Eminent examples of such models include Very Large Instruction Word (VLIW) processors[3] and Explicitly Parallel Instruction Computing (EPIC) which use static scheduling methods and an exposed pipeline[4] to execute instructions. By using such a strategy however, an alteration to the underlying architecture requires compiler amendments, in turn contributing to software compatibility issues (re-compilation, backwards compatibility issues etc.) for newer generation models. Such a compiler model also imposes limitations on instruction execution as general purpose applications have varying degrees of operation-type parallelism. Consequently depending on the application, wide-issue architectures may lead to frequent core underutilization, while also requiring demanding register file ports for such wide-issue execution, contributing to additional power consumption, complexities, and **critical path** latencies[5]. Consequently, a solution is sought to compromise between hardware and compiler directives while maintaining software compatibility

[3]See Chapter 2.8

[4]Exposed pipeline is a technique which provides the compiler with direct knowledge of the underlying architecture, for instance the clock cycle required to perform a given operation, the number of physical registers on core etc.

[5]Latencies which determine a structure's operating frequency

to overcome various ISA limitations.

1.1.4 Single Core Bottlenecks

The following section reflects on further bottlenecks exhibited in more aggressive conventional processor models, namely Out-of-Order (OoO) superscalar models. Such processors support the execution of multiple instructions simultaneously (i.e. superscalar), where instructions may execute once their operands are ready (i.e. OoO). Such a model promotes dynamic and data-flow like instruction execution in an otherwise sequential instruction stream of a program. In order to support such an execution style however, the processor must integrate more complex logic circuits which extract dependencies and guarantee that instructions leave the pipeline in program order for application integrity[6]. Bottlenecks of the OoO superscalar model include increasing issue-width to expose higher Instruction Level Parallelism (ILP, the overlap of instruction execution in a pipeline to improve performance), unscalable and redundant data operand transport, and the cost of mispredictions and exceptions. Such issues are elaborated in the following subsections.

Issue Width and Increasing Instruction Level Parallelism

Considering an OoO superscalar processor, issue width (IW) is the maximum number of instructions that can be issued/executed within the same clock cycle for a given processor core. IW therefore effectively contributes to a processor's maximum attainable ILP. The number of instructions present in a pipeline is also limited by an instruction window, referred to as the Re-Order Buffer (ROB - a structure which holds a list of ordered instructions as they are initiated in a pipeline, allowing instructions to execute OoO and leave the pipeline in-order). Theoretically, increasing the IW and instruction window of a pipeline could potentially increase a processor's attainable performance.

Pipeline structures which lie on a processor's critical path include register renaming and the Issue Queue (IQ), both of which are used to eliminate and monitor operand dependencies, respectively. The bypass network also lies on the critical path and is responsible for forwarding all functional unit (FU) results to their respective consumers which may be present at various stages in the pipeline. Consequently,

[6]Detailed information on superscalar OoO models provided in Chapter 2.4

increasing a core's IW and FUs increases bypass network complexity, in turn impacting frequency [3]. In the case that the bypass network was lengthened, the IQ would also require an increase in capacity to meet the instruction demands of the execution stage, also impacting the number of register file's read/write ports required, which contribute to increasing operand latencies. By increasing the size of the IQ, the fetch bandwidth (i.e initiation, or rather the number of instructions *fetched* per cycle) would also need to increase to provide the processor with instructions to process. Increasing the fetch bandwidth however would especially prove problematic for the complex decoding stage of CISC-based ISAs such as x86[7][3]. The general scalability of conventional processor pipelines therefore exhibits a cumulative effect of drawbacks which inherently limit a processor's ability to increase ILP and sequential performance.

Costly Recovery of Mispredictions/Exceptions

The dynamic execution of conventional processors has led to the implementation and integration of various prediction mechanisms to increase performance, referred to as speculative (instruction) execution. The cost of a misprediction however requires various hardware recovery structures in order to restore the state of the processor previous to the offending instruction. Exception and interrupt handling within a processor follow very similar procedures. The recovery process to handle such events include pipeline flushing, saving and restoring the last non-offending committed state, and in the case of an exception, executing the handler between the saving and restoring process. Due to the latencies incurred for handling such mispredictions and/or exceptions, processors suffer performance loss.

Pipeline complexities for restoring state also contribute to additional core power consumption and area overhead. It is therefore possible that a completely new approach to a processor's model may take these latencies into consideration, while applying alternative and non-intrusive methods to handle such events.

Data Transport and Redundant Bandwidth

Data transport in a processor is the way which a data result (producer) is delivered within the pipeline to its dependent (consumer) instructions. Since processors implement pipelining, its possible that an instruction's dependencies may be

[7]Further discussed in Chapter 2.3

located and/or buffered at various locations/structures in the pipeline. Data transport therefore ensures that a result is forwarded to all its consumers within the pipeline. To achieve such forwarding, more aggressive (superscalar OoO) processor models require that each executed result be written to the register file, and simultaneously broadcasted to the IQ to "wakeup" possible consumer operands. In the case that a consumer was scheduled directly after the executed result, the result is also forwarded on the bypass network directly to its consumer, effectively improving a core's performance.

Studies have demonstrated however that results written to the register file are actually consumed almost immediately within a pipeline [6, 8]. Specifically, 70% of the broadcasts sent to the IQ are unnecessary as they have been forwarded to their single consumer previously on the bypass network, where 74% of the results stored to the register file have already been forwarded to their single consumer and/or never read and/or overwritten by subsequent instructions. 80% of such values also have a lifetime of 32 instructions or fewer [6]. This study therefore emphasizes the redundant bandwidth and unnecessary register reads/writes present in the processor pipeline to accommodate data transport. Accordingly, the elimination of such redundant bandwidth by using a smarter data forwarding process to improve performance, energy efficiency, and register storage allocation with an alternate datapath solution poses as an attractive solution. Grouping related instructions, i.e. coarse-grained, versus the fine-grained execution approach imposed by conventional processors could also contain operand lifetimes in a localized manner to effectively improve the data transport problem.

1.1.5 Multi-Threaded Processors

The inefficiency of single-threaded cores arise when applications exhibit low inherent ILP and/or cache misses [9]. In the case of a cache miss, an instruction may take hundreds of clock cycles to fetch data from the lower levels of the memory hierarchy and return. The processor therefore is not utilized during this time, simply stalling until the instruction is resolved. In the case of low inherent ILP, the number of independent instructions executing simultaneously in the pipeline may not be sufficiently utilizing the pipeline structures, causing underutilization of core resources and unnecessary power consumption. To mitigate such cases, Simultaneous Multi-thread (SMT) processors were proposed as an augmentation to a conventional superscalar,

sharing or duplicating pipeline structures so that multiple threads simultaneously traverse the pipeline to increase core utilization and throughput[10].

Although SMTs were proposed as a way to increase core efficiency, the model's single-thread performance is often sacrificed as thread's scale, especially for more algorithmic- and computationally-intensive workloads. Specifically, certain threads are optimized for independent operations and are intended to fully exploit the core's hardware structures (registers, cache lines and FUs). Consequently when an optimized thread shares core resources with other threads, contention and blocking of pipeline structures arise, causing performance degradation and hindering the benefits of multi-threaded execution[11]. The general sharing of pipeline structures also limits a thread's instruction window, where the IQ and ROB of a fixed size must be divided among the threads in comparison to a single-thread core which may use the entire structure to extract dataflow-like execution. Consequently, the advantages of SMT throughput and utilization as thread's scale are at the expense of individual thread performance loss.

To mitigate thread blocking, larger and/or duplicated hardware structures are integrated into a SMT pipeline, in addition to a fetch policy which invokes an algorithm to mitigate thread contention throughout the pipeline. To support such features and accomodate larger structures, more physical pipeline stages must be added to allow for an acceptable frequency of operation, in turn contributing to an increase in instruction latency, core area and power. SMTs main objective of utilizing a CPU's existing structures has therefore shifted towards the integration of larger hardware in a pre-existing complex pipeline. Consequently, applications requiring more aggressive single-thread performance often resort to alternate models which do not incur such performance penalties, i.e. single-thread core multi-cores.

To provide more insight on the SMT single-thread and throughput issue, Figures 1.3 and 1.4 present select benchmarks of the PARSEC [12] and SPLASH-2 [13] suite. Fig 1.3 displays the average *single-thread* IPC improvement over an equivalent SMT core, considering a multi-core of single-thread cores (STC) and the average IPC attained per core. As seen in Fig. 1.3, single-thread latency increases in SMT processors, where certain threads benefit from executing on multi-core STCs more than others (i.e. approximately 1.34x and 1.95x for 2 and 4 threads, respectively). The greater the IPC gain of the multi-core STC, the higher the level of contention expe-

rienced by the threads in the SMT due to resource sharing and blocking. As seen in the figure, even more performance gains are attained by 4-Thread (4T) workloads for such reasons.

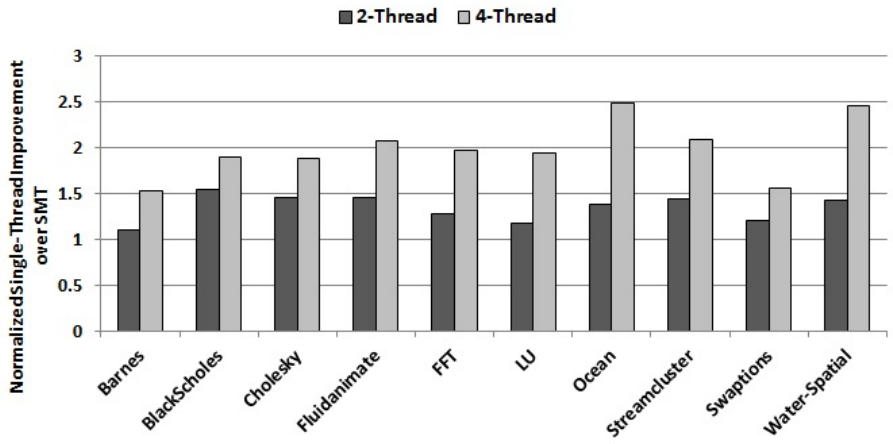


Figure 1.3: Achievable *Single-Thread* Performance Improvement for SMT

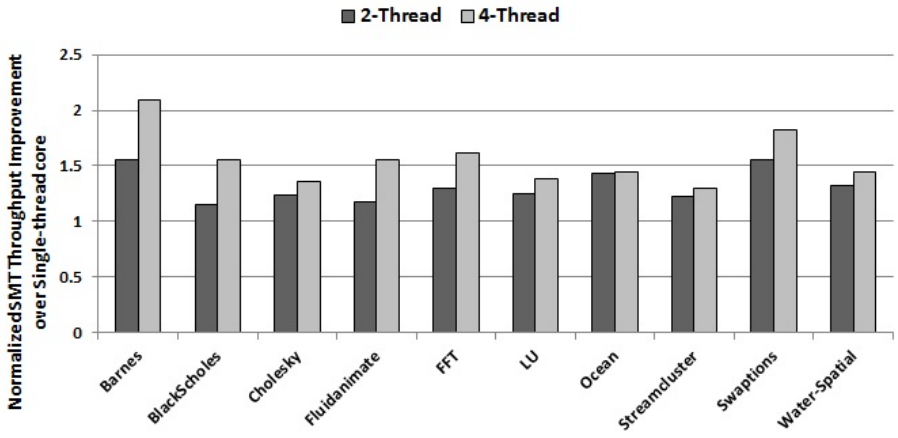


Figure 1.4: SMT Throughput Advantage Over Single-Thread Core

Conversely, Fig. 1.4 displays a SMT's throughput advantage over an average STC. The figure clearly displays that although there is a certain amount of single-thread IPC loss in a SMT core, throughput of the core is in fact improved per core on average by 1.32x and 1.55x for 2 and 4 threads, respectively. This provides motivation for designing a new core which exploits both TLP and single-thread performance concurrently in a scalable manner. This objective is especially important as we continue to proceed in a heterogeneous computing era which compromises

between both ILP and TLP.

Referring back to fetch policies, a major SMT design issue is selecting a policy which mitigates IQ clogging and provides a sufficient instruction window per active thread. Therefore an ideal fetch policy would allow active threads of an application to proceed throughout the pipeline and execute, while stalled threads are not fetched and utilize minimal resources. Such sources for IQ clogging include [14]:

- **Long latency instructions:** Loads which miss in the data cache consume pipeline resources, possibly preventing commits if the instruction is head of the ROB. Such a thread then contributes to IQ clog and limit the instruction window for other threads. Minor clogging issue may also arise in the case of successive and dependent floating point instructions i.e. long latency operations.
- **Long data dependence chains:** If an instruction relies on another instruction with a load miss, all dependent instructions will also stall until the memory load is resolved, causing IQ clogging for other threads.
- **Contention for Functional Units:** Since threads must share pipeline structures, structural hazards may arise (i.e. the unavailability of a resource), contributing to performance loss. Issue width in most cases can not be widened (i.e. including additional FUs) due to the scalability problems discussed previously.

In order to alleviate clogging, many fetch policies have been proposed, along with the concept of runahead threads [11, 15–18]. Such fetch policies aim to reduce resource monopolization based on certain criteria and/or events, often causing an under utilization of resources and/or unintentional thread stalling until certain thresholds are reached [17].

Conversely when the oldest instruction in the pipeline is a Level 2 cache (L2) miss (or locks in the case of multiprocessors), runahead threads allow a conventional pipeline to continue executing instructions using a fake value to represent the load miss. Using the value, the instruction stream continues to execute, allowing future load/store instructions to issue and prefetch data, decreasing the chances of future cache misses. Context must be saved however so that the previous state prior to the fake value is restored once the L2 miss returns. All instructions which execute

after the L2 miss are therefore only required for prefetching and must be squashed (eliminated from the pipeline). Consequently runahead consumes consume additional dynamic power as the pipeline must be restored whether or not the fake value used was correct. Consequently, runahead threads require an increased number of executed/squashed instructions which adversely affect a core and/or SMT's energy efficiency [18]. Such policies and runahead techniques also do not provide the same single-thread performance of single-thread cores.

A recent study conducted by Eyerman and Eckhout compared a homogeneous SMT multi-core system to heterogeneous (non-SMT) multi-core systems for various applications, considering the same power budget per core [19]. The study concluded that *SMT multi-cores generally perform better than heterogeneous multicores* as each SMT core adapts to varying degrees of TLP while providing adequate single-thread performance. Therefore each SMT core was generally able to outperform each optimized heterogeneous core in a certain aspect while eliminating communication latencies and data sharing overhead between threads and cores, typical of heterogeneous systems. As previously discussed however, the SMT model possesses several limitations which may be improved. Therefore if such improvements may be applied to homogeneous systems, more performance gains may be achieved. Such SMT limitations therefore act as the prime motivation of this work.

1.2 Research Objectives

According to the motivation provided in this chapter, the objective of this research work is to develop a processor architecture which provides a nuanced approach to conventional CPUs/CSPPs. The goal of such a core is to provide enhanced single-thread performance for multi-threaded workloads, while eliminating several conventional CPU and SMT model bottlenecks. Accordingly, such a processor should provide more architectural flexibility to efficiently execute and adapt to various degrees of thread-level parallelism (TLP) and programmable procedure workloads, while mitigating the instruction servicing latencies and redundant bandwidth that general purpose architecture impose on conventional pipelines.

The main emphasis of this thesis is therefore to develop a completely new approach to general purpose processor architectures. Primarily, processor performance and the transistor/frequency scaling plateau suggests that industry and academia

must consider computing in an unconventional way. Although computing has steered towards heterogeneous core alternatives, a single core's architecture and performance is essential to performance gain, where a processor's programmable procedure model must be considered.

To accomplish such an objective, several queries were raised which lead to corresponding topics of investigation. These topics were researched and the result of the investigations were used to design the processor presented in this thesis. Such research queries and their respective topics are listed below.

Research Question 1: What nuanced computing model and datapath design may mitigate instruction control and processing overhead to emphasize data processing and increase a single-thread's performance? How can the model be applied to a multi-threaded workload domain to improve upon a SMT model's limitations?

Research Question 2: Is it possible to alter the triple set characteristics of a Computing System with Programmable Procedure (CSPP) in order to mitigate the effects of redundant bandwidth, unscalable structures, and the general bottlenecks raised in Research Question 1? If the triple set is altered, how will this affect the ISA and software/compiler compatibility?

Research Question 3: Is it possible to have a smarter compilation process to extract application characteristics and eliminate various datapath structures? If so, is it possible to maintain compatibility with standard software, ISAs, and programming models? If compatibility is maintained, how will the software interact with the processor's hardware to convey such application characteristics?

Research Question 4: Once the processor is built, how will exceptions and/or mispredictions be handled?

1.3 Thesis Contributions

The main contributions of this thesis are two-fold. The first main contribution is the proposal of a nuanced general purpose, configurable processor architecture designed to improve single- and multi-thread performance. Specifically, to the best

of our knowledge, this is the first general purpose architecture which integrates configurable datapath logic to eliminate various data transport and instruction processing overhead issues in a conventional processor. The design effectively increases the single-thread performance per thread of a given multi-threaded workload, while supporting aggressive speculative execution and a simple misprediction/exception recovery process.

The second main contribution of this thesis is the concept of a compiler compatible process and framework to support such a configurable general purpose processing architecture, referred to as the Physical Compiler (PhysC). Many previous works have attempted to support unconventional underlying architectures through the design of custom primary compilers and/or instruction set extensions. Such approaches however sacrifice software compatibility and are therefore not effective solutions considering general purpose workloads. Conversely, the concept of a PhysC is able to maintain software compatibility by applying a secondary, independent layer to the compilation process. The PhysC may be considered as a *static* and *hybrid* co-designed VM [7] compiler. Specifically, as opposed to translating one ISA instruction to another dynamically, the PhysC performs application macro-processing[8], which follows an application's instruction flow, generating and translating macro-data to configuration logic for execution on the underlying configurable processor. Accordingly, the proposed approach eliminates many front-end bottlenecks, the fine-grained instruction issues of conventional processors, and limitations of co-designed VMs and static compilation, while increasing performance and energy efficiency.

Both contributions were implemented, tested, and compared to conventional processor models using a variety of multi-threaded benchmarks. Results verified that the proposed approach 1) can successfully increase both single-thread performance and TLP on a single chip when compared to both a SMT and single-threaded core model, and 2) achieve such results with up to half the area and 63% of the energy required of conventional models.

The result of this research work is the proposal of a Configurable Simultaneously Single-Threaded (Multi-)Engine Processor (ConSSTEP) which implements the architecture, control, functionality, and software translation process for a nuanced

[8]Macro-processing assembles blocks of code from a program, versus processing individual instructions as typical VMs

multi-threaded processor core. According to the computing system classification presented previously, ConSSTEP revisits the design of conventional processor architectures to rebuild a core, considering the limitations of such systems are in the way which they are designed and programmed, and not necessarily inherent in computing itself. In attempts to redefine the CSPP model and the limitations imposed by ISAs, ConSSTEP proposes both a configurable topology and programmable procedure, such that $A_{ConSSTEP} = \{C_i, \sim L_{i,j}, \sim P_{i,j}\}$. ConSSTEP may therefore be considered as a reconfigurable processor, where the core's topology adapts to a given programmable procedure with fixed (heterogeneous) functional components. Accordingly, such a processor is able to increase single-thread and throughput performance by dynamically configuring and adapting its underlying architecture to the communication patterns of a given workload while eliminating instruction processing overhead. Such configurability is implemented at a coarse instruction granularity to provide minimal overhead for an aggressive, configurable general purpose processor.

1.4 Thesis Statement

ConSSTEP demonstrates that a core's single-thread performance, throughput, and energy efficiency may be improved effectively and efficiently by rethinking the microarchitecture, architecture, and *compilation process* of a conventional processor while maintaining full software compatibility.

1.5 Dissertation Outline

This thesis dissertation is organized as follows: Chapter 2 provides background and details of various conventional processor designs, outlining both their advantages and disadvantages. Based on limitations addressed, Chapter 3 provides a general overview of the ConSSTEP flow and execution process, then outlining the challenges overcome by ConSSTEP in comparison to conventional CPUs in addition to ConSSTEP's trade-offs. Chapter 4 covers various related works which have also deviated from the conventional style of computing, which are directly compared to the ConSSTEP approach.

Next, Chapter 5 provides details of the ConSSTEP architecture, where Chapter 6 details the overall compilation process. Chapter 7 then presents the experimental methodologies used to simulate and emulate the proposed processor and baselines for

performance, power, and area to obtain statistics and analysis. Chapter 8 provides sensitivity studies to determine the ideal scheduling algorithms and engine sizes for the ConSSTEP architecture. Chapter 9 then presents and discusses experimental results. Finally, Chapter 10 provides conclusions based on the findings presented during experimental results, outlining advantages, limitations and possible future work for the ConSSTEP architecture.

Chapter 2

Background

2.1 Introduction

The following chapter provides background information on various conventional CPU models and their respective advantages and limitations. The background here elaborates on the facts discussed during the introduction, while also providing motivation behind the transition of one computing generation to the next. The main objective of such background knowledge is therefore to provide insight on the limitations and challenges posed by conventional models, which has lead to the motivation behind the ConSSTEP architecture.

Accordingly, this chapter first overviews details of the conventional in-order scalar CPU. Thereafter the chapter describes various ISA models, and the need to increase core performance, bringing forth the superscalar Out-of-Order processor. Several of the core's advantages and limitations are discussed, thereafter providing details of parallelism granularity in general purpose applications. The chapter then explains the need for multi-threaded architectures, and computing's transition to the current multi-core generation. Finally, the chapter discusses alternate compute models such as VLIWs, EPIC, and Digital Signal Processors (DSPs), and the concept of co-designed virtual machines (VMs).

2.2 In-order, Single (Scalar) Issue Processors

The conventional **Von Neumann** (VN) computing model can be defined as a store program computer consisting of the following four main blocks [20]:

1. **Central Processing Unit (CPU)-** Responsible for obtaining instructions of a program and processing the data incrementally through a set of pipeline stages. Specifically, the CPU contains a Program Counter (PC) to keep track of the next instruction's address, an on-chip register file for quick access to instruction operands, Arithmetic Logic Unit (ALU) for the arithmetic and logical execution of instructions, a datapath which statically connects all the hardware components to provide pipeline functionality, and a control unit which interprets instructions and the CPU's functional state. Note that the CPU may also contain branching, Floating-Point (FP), complex integer, and load/store units (LSU) for non-arithmetic/logic instructions[1].

2. **Memory -** used to store program instructions, data, and any information pertaining to the CPU and OS, whether intermediate and/or final results. Although memory was implemented as a single block in the original VN model, it is now a memory hierarchy.

3. **Input -** obtains external data and instructions, storing/processing the information for the computer's use through interfacing protocols.

4. **Output -** displays and/or transmits data from the computer to the external environment, also through interface protocols.

The blocks listed above work together to provide the functionality of the VN computing model. As mentioned above, the CPU uses **pipelining** for program execution. Pipelining is a technique used to increase a processor's instruction throughput by dividing the execution of an instruction into several simple, single cycle stages which overlap with successive instructions of an instruction stream [5, 21]. A basic conventional CPU pipeline is presented in Fig. 2.1 consisting of the stages: 1) Instruction Fetch (IF), 2) Instruction Decode (ID), 3) Execute (EXE), 4) Memory (MEM) and 5) Writeback (WB) to the register file [21]. Using a pipeline technique, the execution of the first instruction may be overlapped with the second instruction's decoding, and the third instruction's fetch. Therefore, as opposed to dedicating five clock cycles per instruction for fetch, decode, etc., pipelining effectively increases the processor's performance and ability to process multiple instructions simultaneously for a given programmable procedure.

[1] All units which possess more than arithmetic/logic functionality are referred to as Functional Units (FUs)

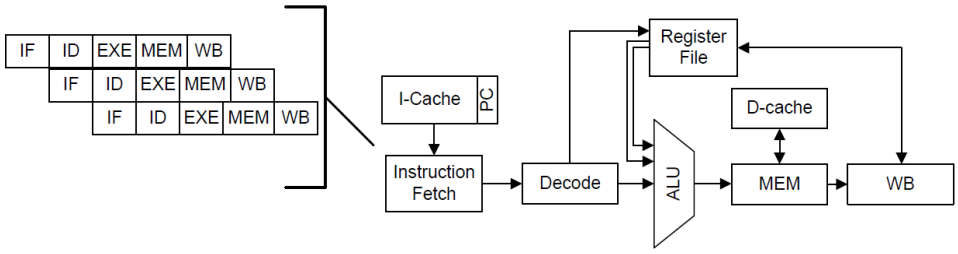


Figure 2.1: Simple In-Order Pipeline

During IF, an instruction is read (fetched) from memory according to the address specified by the PC. The instruction is then latched by a pipeline buffer/register for the next pipeline stage to read, where the PC is also incremented reflecting the next instruction to fetch. In the next clock cycle, the instruction is passed to ID, which decodes the instruction's fields to obtain details including the opcode/operation, input and output operands, and/or addresses for memory or branch instructions, where the instruction metadata is again buffered by the next pipeline latch. Assuming all operands are ready (in-order model), the processor then reads the operands specified by the instruction from the register file and applies them as inputs to the ALU during EXE, which performs a given operation as also specified by the instruction. The result computed by the ALU is obtained and written back to either the register file (WB), or to memory (MEM), or to the PC register for branches depending on the instruction type. In order to maintain program order and correctness, it is also necessary that all instructions and memory accesses are written and read in program order.

2.3 Instruction Set Architectures (ISA)

An Instruction Set Architecture (ISA) is the functional definition of the operations, modes, instruction encoding, and storage supported by the processor hardware [21]. Although the precise implementation of an ISA within a processor architecture may vary depending on vendors and/or processor models, it provides a simplified view of the microarchitecture to the programmer, where each processor supports one target ISA [5]. The role of the ISA is to divide labour between hardware and software [7], allowing for universal software compatibility for various processors and across generations of processor models. Using an ISA, the programmer may design a software program which is compiled and converted to a set of instructions encoded according to a given ISA. The underlying processor which supports the target ISA then ob-

tains the instructions and executes the program. Thus the ISA acts as an interface which allows programmers and computer architects to work independently with the same objective.

ISA's may be classified into three categories: Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), and Application Specific Instruction-set Processors (ASIP). To date, common ISAs aside from x86 include ARM (aarch32 and aarch64), the Itanium series IA-64, SPARC, and PowerPC.

CISC is an instruction set which invokes several complex operations per instruction [5]. Such instructions are of variable length and possess fairly complicated addressing modes. During the 1960s and 70s, CISC was the predominant ISA for computers mainly due to the memory wall – noting that memory access times were much slower than they are today, CISC was able to embed multiple programming constructs into one instruction. Thus CISC required fewer memory accesses while providing code density and performance gains. The overlapped pipelining technique discussed previously however was not originally supported by CISC due to its complex instruction set. Thus as technology scaled and transistors became faster, the RISC ISA emerged in the 1980s, "reducing" the number of instructions, addressing modes, formats and complexity per instruction in comparison to a typical CISC ISA [22].

Accordingly, RISC is an instruction set which possesses one operation per instruction. The ISA employs a general purpose register file (GPR) with fixed length instruction encoding, simple operations, and addressing modes. An instruction's inputs are read from the GPR, where only load/store type instructions are permitted to access memory, contrary to CISC architectures. Due to its formatting, RISC ISAs may support simpler processor designs and control units, where registers allow faster accesses per instruction operand. Although CISC-type processors may also possess GPRs, their operands may be read/written from/to memory or the GPR according to a given instruction's specifications, and hence the architecture must deal with such implementation variability.

Both CISC and RISC ISA's were developed for the general purpose processor domain. However as the embedded computing market continues to thrive, ASIPs have emerged to optimize more catered systems as CISC and RISC ISAs have many fea-

tures which are not necessarily required. Specifically, ASIPs are developed with the intention of accelerating a specific application domain's attainable performance and energy efficiency [23]. Such acceleration is accomplished by developing customized instructions which exploit underlying architectural features for higher application performance. Such ISAs however may only execute a handful of applications as opposed to the generic nature of RISC and CISC. Thus ASIPs have been mostly utilized in Digital Signal Processors (DSP) and Neural Processing Units (NPUs) [23]. Accordingly, the rest of this section focuses on RISC and CISC ISAs for computing systems with programmable procedures.

RISC and CISC each have their individual advantages and limitations. For instance, since RISC employs one operation per instruction, a higher number of instructions are required per application in comparison to CISC. Consequently more instruction memory must be allocated, possibly contributing to higher instruction-cache misses. Conversely, CISC was designed with the objective to complete a task in as few assembly lines as possible. Such instructions however are at the expense of more complex pipeline hardware structures, especially that of instruction decoding. RISC's instruction memory overhead is therefore at the advantage of simpler hardware structures, promoting the concept of *pipelining* which CISC did not initially implement. Specifically, since RISC instructions contain one operation, instructions may execute in a uniform time as opposed to CISC which contain a varying number of operations per instruction. One RISC instruction may therefore be fetched, decoded, executed etc per single clock cycle whereas CISC possesses a variable number of cycles per instruction. To achieve RISC-like simplicity, CISC-based processors invoke a more complex decoding stage which divides instructions into micro-operations (uops) so that the new "instruction"(uop) contains one operation. These uops may then execute with the same pipelining benefits and structures as RISC models.

The pipelining techniques invoked by RISC ISAs have also impacted the memory system of classical computing systems. Specifically, the Von Neumann architecture consists of a single memory which holds both a program's instruction and data. Thus CISC was designed with the concept of such a memory system, using complex instructions for fewer memory accesses, making the ISA inherently slower; CISC ISAs must manage instruction and data accesses from the same memory and so fewer instruction accesses are favourable. In contrast, RISC's pipelined implementation, specifically the IF and MEM stages, require simultaneously access to the

memory system which causes structural hazards as both stages attempt to access the memory system with different intentions (i.e. read/write instructions versus data). Accordingly the *Harvard architecture* was brought forth, separating instruction and data memory so that the two pipeline stages could concurrently access memory without hazards and additional latencies (increased read/write ports etc). As such, RISC may only be implemented as a Harvard architecture, and CISC as both Von Neumann and Harvard.

As RISC and CISC ISAs continue to be predominant in general purpose processors, history dictates that ISAs experience long lifetimes as introducing new instruction sets may raise compatibility issues and disrupt co-operative hardware/software development. Conversely, slow evolving ISAs eventually become a poor match to rapidly changing systems and create gaps between architecture, technology, and possible advancements [24]. Although these traditional means of compiling and computing have more than served their purpose in the past, both ISAs and fine-grained instruction execution have become increasingly confining and partially responsible for stagnating innovation in computer architecture.

2.4 Superscalar/Out-of-Order Models

While in-order pipelining was able to provide adequate scalar processor performance, the model was still bound by a maximum throughput of one instruction per cycle. In order to overcome this limitation, the *superscalar* pipeline was introduced, incorporating multiple ALUs and/or functional units (FU) into the CPU's pipeline (where FUs may perform other operations than the basic arithmetic operations supported by ALUs). The incorporation of multiple ALUs/FUs allowed superscalars to execute several instructions simultaneously per cycle [25]. Processors may also be *Out-of-Order (OoO)*, allowing instructions to execute in a different order from a given program's PC-bounded instruction stream. In this way, instructions may execute once their respective operands and hardware units become available, regardless of program order, the increase overall performance. Such a technique is especially beneficial for long latency instruction, such as cache misses. In such cases, other independent instructions may continue to execute when ready while the miss is resolved, versus the concept of in-order execution which stalls further execution until the instructions is resolved.

Monolithic (aggressive) CPUs combine the methods of superscalar and OoO processing to execute instructions with higher ILP and more of a dataflow-like execution. The Von Neumann/Harvard under linings however require that a program commit its instructions in-order, so that the CPU may maintain memory ordering and precise state in the case of exceptions or other events. As a result, OoO superscalars rely on large centralized hardware resources to support such requirements while simultaneously avoiding data, control, and structural hazards. Centralized structures include register renaming, branch prediction, caches, Issue Queue (IQ), the reorder buffer (ROB), and the bypass network(s).

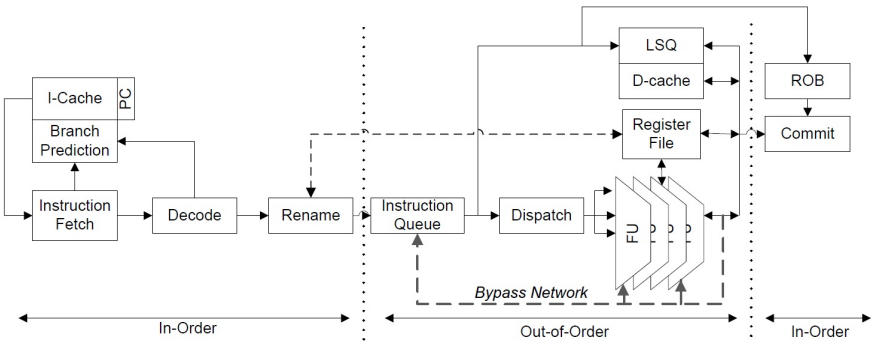


Figure 2.2: Superscalar OoO Pipeline

2.4.1 Architectural Design

Fig. 2.2 illustrates the superscalar OoO pipeline, where its logical structure and operation have remained the same for decades. In the first stages of the pipeline, several instructions may be *fetched* in parallel and *decoded*. Decoding determines the operands and operation(s) required by the instruction. Since the ISA imposes a set of (limited) architectural registers, the instruction source and destination register-based operands are renamed with a pool of the CPU's *physical registers* to eliminate false data dependencies. False dependencies are created due to the lack of registers present in the ISA. Therefore two instructions may write to the same destination register however no actual dependency exists due to the limited architectural registers. Physical registers exist to eliminate such issues, however these registers must still be limited to maintain fast access speeds.

Once renamed, instructions are then dispatched into the *issue queue (IQ)*, and

also placed in the *reorder buffer (ROB)* to maintain the program's order for OoO execution (later used by the *commit stage*). When the IQ and its associative matching logic determine that an instruction's operands are ready and available, the instruction is placed in the *dispatch* queue which awaits structural hazard resolution and an available ALU/FU for *execution* according to the instruction's operation. All operands to be executed by the ALUs/FUs are obtained from the *register file*, or *bypass* network. The bypass network allows computed results to be directly forwarded to their producers, bypassing register latencies so that dependent instructions may execute as soon as possible. Thereafter, once instruction(s) have been dispatched and computed, the instructions await in the ROB to *commit* in program order. Thus when an instruction is at the head of the ROB, it may be committed (i.e. be removed safely from the pipeline).

All instructions are kept in program order at the front of the pipeline until the *rename* stage. All stages prior to the rename stage are referred to as the pipeline's *front-end*. Subsequent pipeline stages are referred to as the *backend* which execute OoO until the commit stage where instructions retire in program order. Conventional coding and compilation strictly rely on these semantics to maintain memory ordering and precise state.

To further improve the performance of an aggressive processor, certain techniques may be used, referred to as speculative execution, and described in the following section.

2.4.2 Speculative Execution

The instruction to be fetched in the next cycle is maintained by the program counter (PC) as previously discussed. A pipelined machine achieves its maximum throughput when in a streaming mode (i.e. continuous fetching of instructions from sequential locations in program memory) [25]. Thus the most ideal of applications would simply increment its PC on every instruction fetch, pointing to the next sequential instruction which is fetched etc. However in actuality, applications possess various conditional statements which redirect control-flow based on a given condition. The ISA expresses such conditions as branch instructions which are processed by the CPU's branch FU. During execution, the branch FU refers to a set of status registers holding conditional flags (i.e zero flags, carry bit set etc) to determine the

outcome of the branch instruction. As an application executes, the status of the bits are updated to reflect state status (for example, the last instruction's result was equal to zero raises a zero flag). Based on the status of the flags, the application's control flow may be redirected to a non-sequential instruction address if a branch instruction is encountered and depending on its type. For instance, if the Branch-if-equal (BEQ) instruction reads that the zero flag is set in the status register, the program fetches from the address specified by the branch, otherwise the next sequential instruction is fetched (where multiple other branch type instructions exist, including BNE, BGT, BLT, etc).

Consider a very simplistic in-order CPU. When a branch is encountered, the outcome of a branch may only be determined once the branch is executed in the backend, an therefore the PC does not know what address to fetch from until the branch is resolved/executed. Therefore once a branch is fetched, the front-end must stall until the branch is resolved thereafter sending the effective address back to the PC so that the CPU may fetch the next specified instruction. In this case, performance is significantly degraded in proportion to the length of the pipeline from IF to EXE. A simple technique to mitigate stalling would be for the compiler to insert (branch) independent instructions after a branch until it is resolved in the backend. When no independent instructions exist, No Operation instructions (NOOPs) may be inserted by the compiler so that the CPU may continue to execute until the branch is resolved, increasing the program size and decreasing a core's energy efficiency. Furthermore, as pipeline stages vary from processor to processor, the number of branch-independent instructions also vary, inevitably leading to pipeline stalls and performance degradation. For this reason, the hardware solution of branch prediction was introduced.

CPUs use branch prediction as a means to anticipate the behaviour of branch instructions so that performance penalties are minimized, and instruction flow throughput is maximized [25]. To achieve such behaviour, branch prediction dynamically speculates branch outcomes (in addition to their target addresses) in the front-end and uses this knowledge to effectively predict a branch outcome, an application's future behaviour, versus waiting until the branch is resolved to fetch the next instruction. To make such predictions, additional hardware logic is placed in the pipeline to speculate and maintain branch histories, where the backend is responsible for performing prediction validation once the branch is executed, performing

misprediction recovery if required. Accordingly, the accuracy of the branch predictor has a significant impact on processor performance.

Common dynamic branch techniques include bi-modal predictors (i.e. 2 bit saturating counters to determine branch outcomes), G-Share (hashing the PC address with history to determine a branch outcome and minimize branch aliasing) [26], and TAGE (a hybrid approach with partial tagging to determine branches with high certainty, using various history tables of different lengths to select the branch direction) [27]. The quality of branch predictors also depends on various parameters such as history lengths, the scope of the history (including nested branches), warmup periods (no solid history of the application is known), and benchmark characteristics.

Branch Prediction and Mispredictions: When a branch is mispredicted, the incorrect path must be discarded from the pipeline. Therefore a pipeline flush is required for all instructions subsequent to the branch, where the processor's state prior to the misprediction must be restored, and the alternate direction must be fetched and replayed. The front-end recovery process also requires the PC to be updated to the correct fetch address, while the incorrect prediction is updated in the history tables, Branch Target Buffer (BTB) i.e. prediction of the branch addresses. Misprediction may also be costly in the backend if all instructions prior to the branch must wait to be committed in the ROB prior to the pipeline flush to maintain program order (especially when considering an OoO superscalar CPU). Thus a low latency technique for branch misprediction is sought. To accomplish such a task, checkpointing mechanisms/structures are used in OoO superscalars. Accordingly, for every branch fetched/decoded, certain pipeline structures are copied or "snapshot" so that the processor state prior to the branch may be used to recover in the case of a misprediction. Such snapshotted structures include the register file, the Register Alias Table (RAT) for register renaming, and the PC, where ROB contents subsequent to the offending branch may simply be cleared using the checkpoint method. Therefore the PC must be updated to the correct address in the predictor, checkpoints must be restored, and the processor state must be rolled back prior to the branch. Once restored, the processor may then resume its normal state and fetch from the corrected path while incrementally re-filling the pipeline.

In terms of the processor's architecture, the greater the number of pipeline stages between fetch to execute, the greater the cost of a misprediction and the recovery

process for re-filling the pipeline with the correct instructions. Likewise as processor architectures and microarchitectures become more aggressive and complex, so does the mispeculation recovery process. Therefore although branch predictors may provide a 90%+ accuracy, the possibility of minor mispredictions still requires checkpointing mechanism integration, with a performance dependency on the number of front-end stages.

Exceptions: It is possible that certain instructions or conditions within the processor and/or its surrounding system may cause exceptions, unexpected changes in control flow causing other instructions within the pipeline to abort prior to their completion [28]. Such exceptional events include IO device requests, OS user services, arithmetic overflow, page faults, memory misalignments or protection violations, hardware malfunctions and power failures.

The listed exceptions may be further classified into two categories: 1) interrupts, and 2) traps. Interrupts are caused by external events (i.e. IOs, page faults, system calls etc) and are asynchronous to program execution. Therefore such external events may be handled between instructions, i.e. after completion of current instruction, where the program may simply resume execution once the exceptional instruction has been injected. Conversely, traps are caused by internal events (i.e. overflows, undefined instructions etc) and are synchronous to program execution. In the case of a trap, the program is suspended, where the exception is remedied by its respective handler. Once the handler has completed, the program resumes its execution (or aborts in certain cases).

When considering the execution of a single instruction, the pipelined processor executes the instruction in segments[28]. In the case of an exception, the offending instruction must always halt execution. Therefore all instructions preceding the exception must complete in the pipeline, where all subsequent instructions are flushed. As the case of mispredictions, the state of the processor must also be saved prior to the exception (including the offending instruction's address, registers, and other structures). The PC is then redirected to the handler's address, where the exception handler is executed. Once complete, the processor restores its state and may continue executing as normal (assuming the exception is recoverable).

Although it is expected that the probability of such exceptional events low, when

exceptions are raised however, the processor must spend time trying to preserve, handle, and restore state to resolve the issue. Consequently, current implementations of exception handling are latency intensive.

Loop unrolling: A **loop** is a method used in programming to repeat the execution of a set of instructions (task) multiple times, based on a number of iterations or until a certain condition is reached. Loop unrolling is a technique used by compilers to replicate a loop's body multiple times assuming independent instructions exist between the iterations, adjusting the loop's termination condition to aid in the overlapping of various instructions in the pipeline. Consequently, loop unrolling takes advantage of ILP in the instruction stream while fully utilizing the FUs of a processor [21]. Loop unrolling is also used to improve instruction scheduling latencies as it may conceal potential branches and overlap independent load and store accesses between iterations.

Loop unrolling is limited however if the overhead amortized by each unroll decreases, yielding insignificant performance gains every iteration. If the unrolling is also performed for a rather large loop, the code size may increase leading to a larger memory footprint, and likely higher instruction cache miss rates which degrades performance. Furthermore, depending on the code size and aggressiveness of the compiler's loop scheduling, register pressure may also increase due to the demand of live registers needed by the processor at a given time. Thus if the processor possesses a shortage of physical registers, although the code is theoretically faster, such unrolling would not be physically possible in the core [21]. Therefore a compiler's unrolling aggressiveness must be effective for increasing performance, and dependent on the underlying architecture.

A type of hardware loop detection may also be accomplished dynamically in the processor. Employed by many Intel CPUs [29], loop detection logic discovers when the CPU is executing a loop. When a loop is detected, branch prediction and the instruction fetch stage are temporarily disabled for the duration of the loop, where the detector injects the loop's instructions (possibly also decoded) to the pipeline. Accordingly, this loop detector unit is used to increase the performance of loops consisting of up to 18 micro-operations (uops), and eliminating 2-3 pipeline stages while maintaining the state of the branch predictor prior to the loop. Such cycle savings performance is at the expense of extra hardware units and constant

loop detection surveillance, increasing the power and area requirements of the core. Accordingly, the general limitations of OoO superscalars are now discussed.

2.4.3 Limitations of OoO Superscalar Processors

Although the OoO execution pipeline is able to substantially increase performance in comparison to a simple in-order scalar core, it does so at the expense of hardware and power consumption. Specifically, the model centers around a sequential instruction fetch stage which introduces control dependencies between all instructions due to a PC-driven approach. The overall ILP achieved by OoO superscalars is also limited by the issue width, instruction window (ROB), IQ size, and the capacity of other hardware units in the pipeline (i.e. physical registers, dispatch queues etc). As a result, all these factors make the conventional processor model fundamentally sequential [30].

Branch mispredictions also require pipeline flushes and complex check pointing mechanisms to recover from mis-speculation, where exceptions and recovery are fairly latency intensive. The associative hardware logic of the IQ which determines ready instructions is also quite demanding and centralized, where only certain issue widths may be sustained before the matching latency and power become significant bottlenecks [3]. Thus we observe superscalars devoting a large share of their hardware and complexity to reconstructing a limited view of program dataflow at runtime [6] where such dependencies are actually known by the compiler and limited by the ISA.

Another major disadvantage of the superscalar pipeline is data movement and transport as discussed in Chapter 1. In the front-end, tags and data needed for matching and forwarding are continuously copied and buffered from one pipeline stage to the next, requiring hardware and interconnection which is often redundant. Likewise, the bypass logic in the backend consists of a complex interconnect, where the greater the number of ALUs/FUs, the more complex the communication infrastructure and the greater the number of register ports required [3]. Furthermore regardless of value bypassing, values must still be written to the register file even if unnecessary, contributing to redundant bandwidth.

Overall, it is evident that superscalar models do not scale well due to vast in-

frastructure of slow networks, associative searches, and complex control logic [7]. It is also clear that the model is very centralized, meaning that there are several single points of control which govern the hardware's execution. This centralization represents a single point of failure, and can often lead to faults disabling an entire core [31]. Thus as technology and transistors continue to scale, reliability must be considered also for die costs and manufacturing [32].

2.5 Parallelism Granularity and Application Flow

Applications may contain up to three types of parallelism: Instruction Level Parallelism (ILP), Data Level Parallelism (DLP), and Thread Level Parallelism (TLP). A processor is typically designed to handle all types of application parallelism, however certain models are more optimized for handling specific types of parallelism in comparison to others. This section highlights the concept and ways which processors address the various types of parallelism. Thereafter, the concept of control-flow and dataflow processors are discussed.

2.5.1 Instruction-Level Parallelism (ILP)

Instruction Level Parallelism (ILP) is the overlap of instruction execution in a pipeline to improve performance [21]. The overall objective of ILP is to reduce execution time by overlapping the execution of as many independent instructions for execution as possible within a given clock cycle. To achieve ideal ILP, superscalars must overcome three issues, namely: an uninterrupted supply of instructions for execution, just-in-time data for the execution of instructions, and the analysis of data dependencies within a window of instructions to initiate concurrent execution of the ready instructions [33].

Typical approaches for exploiting ILP within high performance cores involve employing either a hardware or software approach. In the hardware approach (specifically within superscalars), the IQ is responsible for monitoring operand dependencies dynamically and dispatching the ready instructions [34]. The issue width of the CPU therefore determines how many instructions may execute concurrently. Accordingly, the memory system's bandwidth and latency determines the overlap of instruction data in the pipeline, with the register file providing instruction operands, and the cache supplying memory instructions. Typical OoO superscalar processors aim to

optimize these design constraints to extract higher ILP. Designing wider machines however incurs a geometric complexity increase in control logic, where issue width is also bounded by bypass scalability, and the L1 cache's bandwidth. [28] [3].

In the software approach, the compiler relies on software functionality to find instruction parallelism statically during application compilation. The compiler design also strictly depends on the underlying architecture, its issue width, and an exposed pipeline. Consequently this technique raises issues of binary and software compatibility as a compiler must support multiple underlying architectures and processor generations, versus superscalars which support dynamic scheduling regardless of its architecture. Furthermore, certain applications may possess limited ILP and therefore providing a processor with a very wide instruction width may not be beneficial, leading to an under utilization of processor resources. This technique is mainly used by VLIW and EPIC processors, further discussed in Chapter 2.8.

2.5.2 Thread-Level Parallelism (TLP)

Thread-Level Parallelism is a processor's ability to execute independent programs and/or thread contexts simultaneously on a single core [21]. TLP models allow multiple threads to traverse the pipeline in parallel, giving the illusion of a multi-tasking processor. Such TLP and/or multi-tasking relies on the OS which creates and overlaps multiple threads of execution for the CPU to process concurrently. By supporting TLP, a processor may improve its core throughput to conceal various latencies (in comparison to executing a single-thread workload). Such TLP processor models include Simultaneous Multi-Threading (SMT), Fine-grained MT, and Coarse-grained MT (described in the *Single-Thread vs Multi-Threading* section of this chapter).

2.5.3 Data-Level Parallelism (DLP)

Data-Level Parallelism (DLP) is the ability for multiple threads of execution to perform identical operations on different data sets simultaneously [21]. The technique of DLP involves exploiting parallelism using data streams, whereas conventional ILP/TLP aim to exploit parallelism for multiple instructions using the same data stream. CPUs exploit DLP through Single Instruction Multiple Data (SIMD) operations, which are executed on dedicated backend SIMD FUs. SIMD operations behave as vector processing units, where the same instruction is executed on a row

of data elements simultaneously. Such DLP is exploited in the source code statically, where performance is highly dependent on the programmer's ability to express such computations as parallel vectors, and for the compilers to detect such operations.

To exploit DLP operations, ISAs are often augmented with SIMD instruction set extensions. SIMD extensions were originally introduced ISAs for computing multimedia-based applications, i.e. MMX (128b vectors), but were soon also applied to scientific applications. Such SIMD extensions include SSE (256b) and AVX (512b) for the x86 ISA, with NEON for the ARM ISA. Accordingly, vector lengths vary between architectures and ISA extensions. Consequently software may need to be re-compiled or even rewritten for portability and compatibility between various processor platforms of different vector lengths [35]. Thus DLP issues remain beyond the scope of this work.

2.5.4 Control-Flow vs Dataflow

The processors discussed in this chapter, also referred to as *Control-Flow* based processors, use a memory system to store instructions and data, where instruction flow is driven by a program counter (PC). A program however is an implementation of an algorithm, which can be represented as a dataflow graph. Such dataflow graphs consists of vertices and edges which represent instructions and their dependencies, respectively. Accordingly, complex pipeline structures such as the IQ are used to reconstruct a limited view of this program dataflow at runtime.

To extract such program dataflow, many works have proposed pure dataflow processors, which require dedicated programming languages to represent a given algorithm as a dataflow graph[36, 37]. In *dataflow* processor systems, the execution of an instruction is triggered purely by the availability of an instruction's input operand data. Triggers are implemented using tokens, where an instruction may execute when all its input tokens are present. Once executed the result token is distributed to its consumers, thereafter triggering other ready dependent instructions. Tokens therefore contain such metadata and independently traverse the pipeline to trigger functionality. Such a design eliminates the need for a PC and control-flow processing execution style, where data results may be passed directly from producer to consumers without the need for complex logic structures as the case of sequen-

tial instruction streams. Instructions in data-flow processors may also be discarded from the datapath/pipeline once executed, versus the ROB method employed by OoO superscalars.

Accordingly, the success of dataflow machines have heavily relied on the following requirements: 1) the ability to support imperative languages, 2) the efficiency of executing *hybrid control-flow/dataflow* based operations, 3) the need to manage data structures, memory organization, loops, and sub-functions in hybrid models and 4) the importance of understanding static and dynamically allocated instruction placement, tokens etc. Such dataflow processors often require custom compilers and ISA enhancements to support such custom languages. These architectures therefore cause software compatibility issues in modern programming and computing. For these reasons, control-flow machines have remained prominent. However recent works have invoked hybrid control and data-flow techniques to propose ways to compromise between the advantages of both processor approaches. Such works are discussed in Chapter 4.2.

Next, this chapter discusses details of multi-threading for increasing performance and better utilizing a given processor core.

2.6 Single-Thread vs Multi-Threading

A conventional single-thread CPU executes one thread per pipeline. As previously mentioned, in the case of interrupt, exception or OS call, the pipeline state must be saved, flushed, where an alternate thread then starts/restores its state and executes. Similarly for cache misses and branch mispredictions, a single-threaded pipeline may be stalled for several cycles until resolved, leading to an under utilization of hardware. Therefore although such events are application dependent, single-threaded cores limit the maximum instruction throughput of CPUs. This leads us to the concept of executing multiple thread contexts to increase core ILP and TLP. To accommodate multiple thread contexts, a single CPU core may be amended to support different granularities of Multi-Threading (MT). The following describes the three main techniques employed by processors for achieving multi-threading (MT) on a core: Fine-grained, coarse-grained, and simultaneous.

2.6.1 Fine-Grained Multi-Threading

Fine-Grained MT processors fetch instructions from a different thread per clock cycle, such that every pipeline stage possesses a different thread's instructions. This technique improves pipeline utilization while tolerating control and data latencies, yielding sufficient throughput by overlapping threads with useful work [38]. Instruction/data dependency mechanisms and branch prediction are typically not required as threads do not overlap in the pipeline. Although this may be an effective technique for hiding latency, single thread performance suffers greatly since one instruction fetch is performed per thread every P cycles (where P is the total pipeline stages). Hardware complexities, selection logic, and certain resource contentions are also problematic.

2.6.2 Coarse-Grained Multi-Threading

Coarse-Grained MT processors execute a single thread context in the processor until a stalling event occurs. The pipeline's state is then saved and flushed on such events, where a different (ready) thread is selected to run in the pipeline. The new thread is then restored to its previous state and executed. Such context switches however incur latency overhead, where deeper pipelines are subjected to suffer higher performance losses [37]. Although this method is much easier to implement in comparison to other MT techniques, coarse-grained MT suffers from low single-thread performance while yielding non-deterministic delays and performance.

2.6.3 Simultaneous Multi-Threading (SMT)

Similar to coarse- and fine-grained MT processors, SMT also fetches from one thread per cycle, however allows multiple threads to dispatch and execute their instructions concurrently on a given core [15]. Such a technique is supported through the sharing and duplication of multiple pipeline structures, allowing for highly utilized pipeline stages and the execution of independent operations from various thread workloads. As some structures are shared, tags are also necessary for distinguishing thread contexts and their respective dependencies. SMT therefore requires additional complexities within a processor for better utilizing the datapath and providing higher throughput.

Diminishing performance returns in SMT CPUs remain an issue as the proces-

sor scales to support more thread contexts. Such scalability limitations are due to blocking, contention, structural hazards, potentially longer pipelines, and thread over utilization. Accordingly commercial CPUs do not exceed 2 threads per core for more strict memory models (i.e. x86[2]), and up to 8 threads for relaxed memory models (i.e. SPARC etc). Similar to other MT techniques, single-thread performance also degrades as threads scale due to thread selection, resource sharing and blocking. Consequently, the amount of threads executing per core must be restricted to trade-off performance and hardware constraints, resource contention, and the overall instruction parallelism found within a general purpose workload. SMT support for embedded CPUs however has not been implemented since a slight increase in performance for embedded workloads is at the expense of energy consumption and extra hardware logic.

Although SMTs have both their advantages and disadvantages, multi-threaded workloads may also be deployed on multi-core models, described next.

2.7 Multi-Core Models

Technological innovation stems from the principles of Moore's Law and Dennard scaling. Dennard scaling suggests that maintaining constant electric fields and reducing transistor dimensions by 30% every two years will allow MOSFETs to consume less power and maintain reliability [1]. Thus Dennard forms the basis of Moore's Law: as transistors continue to scale, the number of transistors within a chip will double approximately every two years.

Transistor scaling has accounted for almost three-orders of magnitude in performance improvement over the past 20 years of processor history [1]. According to Dennard's scaling, as transistors are scaled by 30%, their area reduces by 50% to double the density of every technological generation. Scaling also reduces a transistor's delay by approximately 0.7x, conversely increasing frequency by 1.4x [1]. Furthermore, by keeping the electric field constant, supply voltage (V_{dd}) can be reduced by 30%, in turn reducing power by 50%[3]. Chip architects have been successful at increasing frequency and creating complex architectures within a power budget by exploiting these transistor trends.

[2]Referred to as Hyper Threading (HT) by Intel

[3]Since $P_{dyn} = \alpha CV_{dd}^2 f$, assuming a constant activity factor α

As presented in the Introduction chapter of this thesis, Fig. 1.2 outlined CPU trends from the 1970's until 2016, including the number of transistors, frequency of operation, and the thermal design power per microprocessor for more eminent architectures. Similarly Fig. 1.1 displayed the architectural advances and energy efficiency of processor styles. Specifically, Fig. 1.2 displays that beginning in the early 2000's, transistor counts continued to increase while the effects of clock speed, power, and performance concurrently became elusive. As a result, the CPU scaling trend no longer followed Moore's law or Dennard's scaling. In order to restore such parallelism, performance, and the benefits of CPU scaling, multicore architectures were introduced around the mid 2000's. As seen in Figure 1.1, these multicore (non-deep pipelining) methods were able to revive performance and power scaling by liberating processors of their expensive and inefficient techniques, towards a simpler means of extracting parallelism. It is evident however that the last few multicore generations have provided a performance/watt plateau as displayed in Fig. 1.1.

2.7.1 Motivation

The main concept behind multicore systems as demonstrated in Fig. 1.1 and 1.2 is to trade design complexity for power/area efficiency. Accordingly, symmetric (homogeneous) multicore (SM) architectures employed often adopt one of two design methodologies: connect a large number of small, simple in-order cores to deliver high thread-level parallelism and power efficiency, or a smaller number of OoO SMT-based superscalars to negotiate between single- and multi-threaded performance [39]. Additional hardware units are also necessary for symmetric multicores to maintain coherency, communication, and memory semantics among the cores.

Although certain SM's may tolerate low-power in-order cores by compensating low single-thread performance for an abundance of thread-level parallelism, general purpose systems still require the single-thread performance and high ILP provided by aggressive, OoO processors [40, 41]. In order to directly compensate for the lack of single-thread performance in symmetric multicore systems, heterogeneous multicores were proposed [38]. These systems contain cores of various sizes, performance, and complexity to compromise between single- and multi-thread performance. They typically consist of few aggressive cores for sequential code execution and other core models (i.e. SMT, simple in-order) which are dedicated to increasing on-chip

throughput and TLP. Although results suggest that the cores are able to speed up sequential applications in comparison to a purely symmetric approach, several issues such as scheduling, core configuration, uniform ISAs, and utilization remain an area of research [1, 42]. Successful implementations of heterogeneous MP include ARM's big.LITTLE processor series.

2.7.2 Limitations

Since 2005, processor designers have steered towards increasing core counts to exploit Moore's Law rather than focusing on the importance of single-thread performance. As seen in Fig. 1.1, non-deep pipelines in conjunction with a multicore architecture obtained a 3.5x performance/watt average improvement in comparison to the previous generation. However slow performance improvements have been observed thereafter as chip architects must limit the frequency and number of cores to keep power within reasonable limits [1]. It is also imperative that a certain performance gain is acquired with every advancing computing generation despite the decline of transistor and frequency scaling [43]. Majority of multicore hardware research however has been directed towards energy efficiency and specialization.

The work of Esmaeilzadeh et al [43] assess Dennard's scaling and various other characteristics to predict the number of future multicore scaling generations we are to expect. Combined with ITRS's optimistic projections, the study predicts that the best average speedup attainable from multicore systems will be 16% per year until 2024 when the 8nm technology is reached [43]. This figure is fairly accurate given that Intel's Haswell processor only provides a 14% performance improvement (on average) when compared to its predecessor.

Based on these trends it is evident that the performance gains and energy efficiency we have grown accustomed to cannot be achieved through advances in conventional architectures and simply relying on transistor scaling. Whether these trends terminate due to energy/performance scaling issues or Moore's law remains unknown. Regardless, the goal of delivering performance and compatibility with every advancing computing generation will remain a constant goal for architects. To continue these trends, it is important that technological dependencies are mitigated by researching novel architectures which diverge from the standard model of computing. This thesis now briefly discusses other well-known alternative computing

models.

2.8 Very Long Instruction Word (VLIW)

VLIW processors move the complexity of the hardware to the compiler, providing more of an intuitive compilation process. Using this technique, the compiler may extract higher ILP from a sequential instruction stream using a wider issue width. VLIW architectures exploit such operation parallelism by positioning ALUs/FUs of specific types horizontally in the processor. Very long instruction words are created statically by the compiler according to the pipeline of the underlying architecture. The processor then fetches these long instructions consisting of multiple operations which must simply be decoded and executed, effectively reducing front-end bandwidth (i.e. fetch, decode, rename etc). A demanding amount of memory however is required to store such wide instructions.

As VLIWs are more compiler intensive, the CPU's hardware and logic are also much simpler in design, exhibiting lower power consumption and on-chip real estate in comparison to superscalar OoO models. In addition, VLIWs do not require hazard detection or hardware units such as register renaming or ROBs since all issues are resolved by the compiler through *static scheduling*. Although VLIWs are simpler in design and obtain a higher frequency of operation, the ingenuity of superscalar designs were able to quickly match such performance gains. Examples of VLIW processors include Intel's i860 and Transmeta's Crusoe [44].

VLIWs ideally depend on an abundance of application ILP and operation parallelism to exploit their full potential. However, if insufficient application parallelism is present, the architecture hardware units are underutilized and unscheduled slots translate to wasted power and hardware. To address such limitations, techniques such as loop unrolling have been used for aggressive execution in VLIWs at the cost of increased code size and additional memory storage. VLIW binary compatibility also poses as a problem, since the compiler is designed based on an exposed pipeline (i.e. the pipeline specifications and latencies of the ALUs/FUs present in the processor). Different VLIW processor implementations may consist of different microarchitectures per generation, and hence maintaining binary compatibility for software across various models becomes problematic. Therefore VLIW advantages are at the expense of a complicated compiler design, and the loss of a pure dynamic

issue/execution method as the case of superscalar OoO processors.

OoO execution allows a core to dynamically determine its operand status, versus VLIWs which assume and schedule instructions based on the compiler's knowledge of the microarchitecture. Thus if the instruction takes longer than anticipated, i.e. a cache miss, the processor must stall as the compiler considered a certain latency for the cache access; the compiler can not predict the outcome of a branch. Conversely OoO superscalars may schedule and execute other independent instructions in such cases, effectively increasing performance. The dynamic approach of the OoO superscalar therefore allows compatibility across processor generations with generic compilation processes which is independent of the underlying architecture unlike the static compilation.

2.9 Explicitly Parallel Instruction Processors (EPIC)

EPIC architectures extend many VLIW concepts by integrating both a static and dynamic approach for handling instruction streams. Although EPIC processors are statically scheduled, they require extensive software/hardware assistance to provide the processor with information to deal with events such as branch prediction, load speculation, and necessary exception handling. As a result EPIC architectures require mechanisms to communicate this parallelism to its underlying hardware [45].

Many measures have been taken by EPIC-based architectures (such as the Itanium IA-64) to mitigate the effects of VLIW processors. For instance, decreased code sizes are managed by compressed instruction storage [20] and software pipelining [46]. To address binary compatibility, processors such as the IA-64 provide extensions for aggressive software speculation which overcome hardware dependency limitations [21]. For branching, methods such as predicated branch execution and trace scheduling have been employed. Latency and memory operations also remain problematic, with possible solutions including software prefetching and cache hierarchy storage prediction [46, 47]. Consequently, although EPIC models solve certain VLIW limitations, they also have much room for improvement.

2.10 Digital Signal Processors (DSP)

DSPs are considered special type of processors used in embedded applications for digital signal processing. DSP hardware units are somewhat diverse from traditional general purpose processors, consisting of special purpose hardware to directly deal with signal processing. For instance, a DSP will often be chosen over other embedded processors if an application is in extensive need for a Multiply-Accumulate (MAC) unit - a unit capable of executing a multiplication and addition/accumulation as one instruction [21]. DSPs are also used for other dedicated communication algorithm accelerators.

DSPs are often employed for fixed-point calculations and accordingly possess extra wide registers that guard against rounding errors. Word sizes in DSPs are also not restricted to power of 2 sizes as in the case of general processors. Thus DSPs, such as the Motorola DSP56301, possess a 24-bit data width and 56-bit accumulator width, versus a typical processor width of 32-bit, 64-bit, etc. For many of these reasons, DSPs remain predominant mostly in the embedded industry for signal processing.

2.11 Co-Designed Virtual Machines

As demonstrated throughout this chapter, a processor's hardware has architecturally transitioned per computing generation, especially when compared to the original logic captured by general purpose ISAs. To minimize these interfacing effects and promote the role of the processor, the concept of *virtualization* is used. Virtualization places an abstraction layer between resources and the user of the resources so that the logical view of the system is different from the physical view[7]. This concept brings forth Virtual machines (VMs) which virtualize a full set of hardware resources, i.e. a processor, memory, storage and peripheral devices, and emulate them on another underlying system. Specifically, VMs may be implemented as individual process and/or complete OS environments. Such virtualization therefore allows for program to architecture flexibility.

Co-designed VM technologies however differ from VMs as their prime goal is to provide intrinsic compatibility at the ISA level [7]. Co-design VMs relieve ISA impositions by virtualizing one processor's instruction set to another. Specifically,

VM software is used to maintain software compatibility, translating a program's instructions of a given target ISA to another (possibly new) ISA which is supported by an alternate underlying processor. Such a process therefore translates virtual instructions to real instructions so that they may be understood by the underlying architecture. This method is primarily invoked to enhance and/or remove functionality from the catered underlying architecture to increase a processor's performance and/or energy efficiency. It is therefore required that co-designed VM software be designed concurrently to a processor's architecture to promote such ISA flexibility and software compatibility.

Eminent works of co-designed VM implementations include Transmeta's Crusoe[44] and Nvidia's Denver [48]. These works have attempted to implement runtime (dynamic) co-designed VMs to accelerate performance on-chip at the trade-off of software complexity. Both works implemented software compatible VLIW pipelines to support wide issue execution, aiming to simplify the underlying hardware and maintain software compatibility for VLIW processors. To mitigate runtime software overhead, these co-designed VM cores include translation caches which hold previously translated (recurring) functions to speedup execution[48]. Consequently non-previously decoded phases suffer a decoding penalty to support dynamic software translation process especially when compared to hardware execution. To decrease this decoding overhead, processors such as Denver multiplex the VM with a hardware-based decoding stage so that previously non-translated code phases may be redirected to hardware, effectively decreasing software overhead [48].

Co-designed VMs are a viable solution for enabling new hardware architecture and techniques which may otherwise be limited by ISAs and prohibited by backwards compatibility. Although there are certain difficulties to the approach, more research emphasis would prove promising for increasing performance of software/hardware in processor architectures.

2.12 Summary

This chapter provided detailed information pertaining to various factors within processor computing. The background provided in this chapter illustrated various conventional processor models, discussed the concept of instruction sets, and demonstrated the general need for the improvement, scalability, and innovation within cur-

rent generation processors. Accordingly, the need to design unconventional models which completely deviate from conventional models poses as an attractive solution. This thesis now discusses and overviews the proposed architecture design of this thesis.

Chapter 3

ConSSTEP Overview

3.1 Introduction

Now that sufficient background knowledge has been provided on processor architectures, this chapter provides a general overview of the thesis' proposed approach – ConSSTEP, outlining the core's compilation process and architecture. Specifically, this chapter provides an overview of the ConSSTEP core, detailing any assumptions made throughout the processor's design. Thereafter the general compilation process is described, along with the microarchitectural and architectural flow for executing the compiled data, where an example of ConSSTEP's execution flow is then provided. Thereafter, a brief discussion on the advantages of the ConSSTEP approach is provided, and directly compared to the limitations of conventional processor methods, structures, and pipeline stages, in addition to the static compilation approach of VLIWs. Finally, although ConSSTEP presents many advantages such is accomplished at certain trade-offs, discussed in detail during the final section of this chapter.

3.2 General Overview

The overall ConSSTEP flow is presented in Fig. 3.1. ConSSTEP is a *multi-threaded* configurable processor consisting of multiple *engines*. Each engine possesses a varying number and type of functional units (FUs) connected through a *registerSwitch* (rS) interconnect. *Each engine* is dedicated to a *single* thread's workload using a low complexity and scalable logic design. Multiple engines work simultaneously on a core to provide high throughput and TLP flexibility for multi-threaded applica-

tions. FUs present in the engine may possess varying integer-based functionality ranging from simple ALU operations, barrel shifting, to complex integer execution, and hence are referred to as FUs.

An engine's components execute configuration data as opposed to ISA based instructions. Engines mitigate much of the overhead associated with reconfigurability by compromising between CSPP and RCS approaches to increase performance. Such configuration logic is generated through a two-layer compilation process. This layered approach is used to maintain compatibility with current software while catering to an underlying configurable architecture. The first layer of compilation is referred to as **Logical** compilation (i.e. the standard compilation process), and the second stage referred to as **Physical** compilation (i.e configuration data generation).

The overall goal of a PhysC is to provide the advantages of a smarter compilation process[1] and the software compatibility of co-designed VMs [44, 48] with minimal runtime overhead. A PhysC's main objective is to obtain a compiled binary and perform macro-processing, generating bundles of microcode logic so that a general purpose application may execute on an underlying configurable architecture. The PhysC's approach eliminates software compatibility issues such as those experienced by VLIWs and VMs by co-designing the PhysC with the processor architecture, however taking a pure hardware approach during runtime. Accordingly, a coarse-grained macro-processing approach is used to translate data/instructions versus co-designed VMs which simply convert one ISA to another (fine-grained method), incurring high software overhead. Instructions are decoded, renamed, ordered, and translated to configuration logic, to eliminate the runtime bottleneck of VMs and execute purely on hardware. The ConSSTEP architecture also completely deviates from conventional models by using such configuration logic, and is able to adapt to general purpose workloads, eliminating many VLIW and superscalar impositions through a new datapath design. By using such a layered approach, ConSSTEP is able to modify internal microcode between processor generations with full software compatibility.

[1] Referring to the static and intuitive compilation processes of VLIWs and EPICs

3.3 Assumptions

The following work and methods assume a Unix-based file system and RISC target ISA. This approach allows the PhysC to implement a simple instruction decoding phase, with fixed length instructions and one operation per instruction. The PhysC flow also assumes that a given benchmark may be translated to instructions using any compiler, however the methodology elaborated upon in this work assumes a gcc/g++ compiler for a target RISC ISA. The architecture also assumes precise exception handling. Although a PhysC and ConSSTEP core may be designed and scaled to any Operating Systems (OS), compilers, and ISAs, such implementation remains future work.

It is also assumed that benchmark applications may be of single and/or multi-threaded programming models such as OpenMP [49] and PThreads. As the main objective of this work is to increase single-thread performance within multi-threaded workloads, such programming models must be considered. In the case of OpenMP, the thread parallelizing model is initiated from a master thread which spawns (i.e. forks) slave threads when a given task in the program is specified as parallelizable. A task is specified as parallelizable using "pragma" keyword statements embedded by the programmer into the code[49]. The OpenMP model therefore takes a higher-level approach to coding multi-threading applications, where the master thread forks threads during runtime at pragma locations so that threads may execute concurrently. Threads then join back to the master thread once the parallelized code has been executed using barriers (i.e. thread wait statements which synchronize all threads).

PThreads conversely follow a more lower-level model with complex programming constructs, requiring explicit programming effort at a fine-grained instruction level. Accordingly, the programmer must specify forks, joins, mutexes, and signalling between the threads and ensure program correctness. Due to such a low-level multi-threading approach, Pthreads are invoked in the C programming language, whereas the higher-level OpenMP model may be applied to a wider variety of languages (C, C++, Fortran). Accordingly, ConSSTEP and the PhysC support all such languages and multi-threaded models.

Finally, considering other programming models such as OpenCL, CUDA etc

which integrate various processing models within a heterogeneous core (i.e. GPUs, accelerators etc), ConSSTEP may be fully supported. Specifically, ConSSTEP was designed as a power and performance efficient substitute to a SMT model. Since the heterogeneous compilation process extracts program phases and compiles them according to the core which it must be redirected to, ConSSTEP may be fully supported. That is, ConSSTEP is fully compatible with conventional single- and multi-threaded ISAs and compilers, and therefore may retrieve the binary which is generated, redirecting it for processing to the PhysC. Since ConSSTEP also supports barriers and memory coherency in a similar manner as conventional CPUs, no issues arise for core integration within a heterogeneous system.

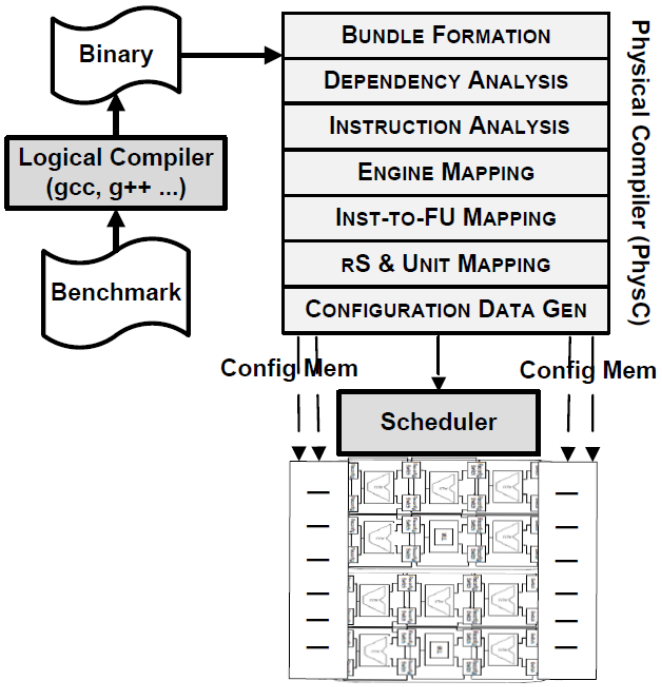


Figure 3.1: ConSSTEP Top-Level Overview

3.4 Compilation Process

The left of Fig. 3.1 illustrates a standard compilation process. That is, a benchmark is input to the logical compiler (i.e. any standard compiler) to generate a binary. Precompiled binaries therefore do not require any recompilation assuming the next step, i.e. Physical Compilation (PhysC) also supports the same target ISA.

In order to generate data for the underlying configurable architecture, the binary is sent to the PhysC for further processing. With the binary, the PhysC is able to use its embedded microarchitectural information to 1) gather instructions to form bundles (terminated on conditional branches), 2) perform instruction analysis within a bundle to eliminate internal data hazards, which is then used to 3) extract intra- and inter-bundle dependencies, where duplicate bundles are removed, 4) select the most suitable execution engine per thread, assigning individual instructions to an engine's internal FUs, 5) determine the data transport of operands with respect to their dependencies (rS interconnect, storage, propagation etc), and finally 6) generate configuration data for the engines and their respective configurable components to execute the given thread workloads.

Once generated, configuration data is routed to its respective engine and configuration memory bank (Fig. 3.2), where bundle addresses and dependencies are provided to the scheduler. Alternatively, scheduling may also be handled by the OS. However since this work's objective is to maintain software-based compatibility, it assumes an on-chip hardware scheduler where threads are managed by the OS.

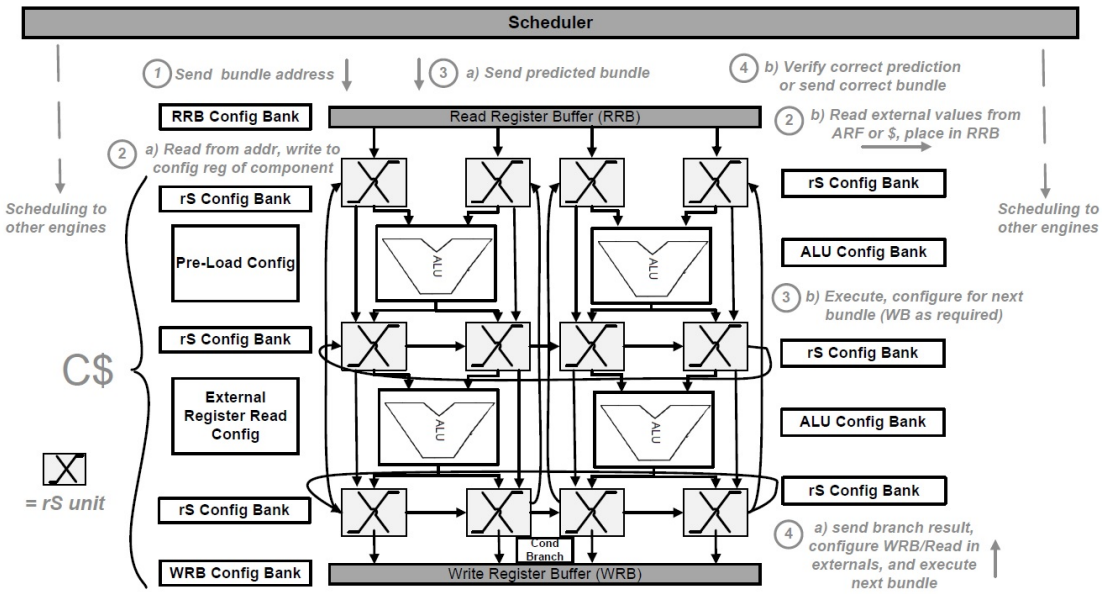


Figure 3.2: ConSSTEP Execution Process (Single Engine)

3.5 Architectural Flow and Pipeline

As previously mentioned, ConSSTEP is a general purpose multi-threaded processor consisting of variable sized engines, where each engine comprises of FUs interconnected through a registerSwitch (rS) network, shown in Fig. 3.2. The rS interconnect is a *configurable topology* where rS units are programmed by the generated configuration logic, dictating the topology's communication characteristics for a workload. That is, an rS interconnect may be programmed to provide *distributed storage* and/or *single-cycle data propagation* between dependent instructions per clock cycle. Accordingly, an rS interconnect allows an engine to temporally configure itself per cycle to support various data transfers and storages, adapting to a given thread's execution workload. A simple example of an rS unit's architecture for the latching and/or propagation of data (consisting of a single input and output port) is displayed in Fig. 3.3. The configuration registers shown are loaded with the PhysC's generated bits, in turn dictating the control logic for a given rS port at a given cycle. A given input port may temporarily store and/or propagate a value as required, mitigating unnecessary buffering and/or data movement (further details provided in Chapter 5.2). The FU functionality itself is not configurable, however the PhysC must determine each FUs operation per cycle according to the workload.

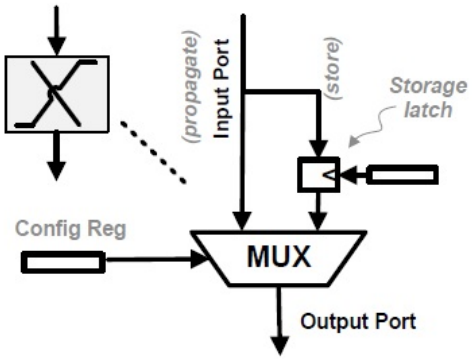


Figure 3.3: rS Architectural Functionality (Single Input and Output Port)

Each engine also possesses a small external architectural register file (EARF) to maintain software compatibility and the basic registers required of the target ISA. Specifically, the rS interconnect is responsible for managing propagation and temporary storage among instruction within a bundle, i.e. *intra*-bundle dependencies, without the need of register file intervention. However a thread consists of

many bundles, where a bundle's final operands may be input to the next (dependent) bundle scheduled afterwards i.e. *inter*-bundle dependencies. To satisfy such inter-bundle dependencies, any final architectural register values present within a bundle (operand results which are not required further by instruction consumers) are written to the EARF, where subsequent bundles may use such data to satisfy the inter-bundle dependencies of a thread. However since the rS interconnect is able to maintain majority of operand lifetime within an engine/bundle, reads and writes to the EARF are decreased significantly.

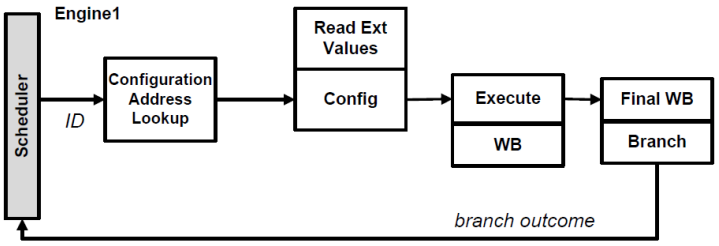


Figure 3.4: ConSSTEP Basic Pipeline

ConSSTEP's basic pipeline is shown in Fig. 3.4. As seen in the figure, ConSSTEP eliminates majority of conventional pipeline stages and places more emphasis on data processing. According to the pipeline's first stage, the scheduler sends out a bundle ID for execution which is read from the configuration memory. In the second stage, the engine is configured while any input data is read from its respective location (EARF or cache). The third phase executes the bundle while performing any writebacks, where the final pipeline stage is responsible for final writebacks and providing the scheduler with the branch outcome. Using the outcome, the scheduler then determines the next bundle to execute, where the respective bundle ID's sent out by the scheduler for configuration and pipeline flow continues. It is also worth mentioning that although the configuration and execution pipeline stages are presented as a single pipeline stage, each stage is inherently pipelined and takes several cycles to complete (i.e. configuration may take up to 10 clock cycles, execution of a bundle varies etc). Each stage however is represented as one in the pipeline figure as they are considered the same pipeline phase.

Although the presented pipeline eliminates many conventional front-end stages and logic, it is not ideal as the clock cycles attained by eliminating such processing overhead have been redirected to the cycles required for engine configuration. Con-

sequently very minor performance gains, if any, would be expected of ConSSTEP with such a pipeline. A more aggressive technique is therefore sought to conceal such latencies and exceed the performance of current SMT and monolithic OoO superscalar cores.

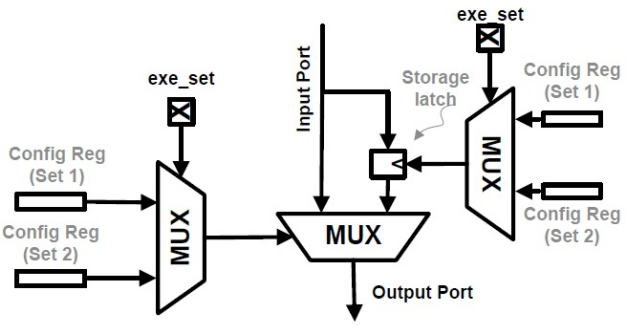


Figure 3.5: rS Unit - Double Configuration Register Setup

To adequately conceal configuration overhead, a two configuration register setup is used by ConSSTEP's engine structures. The idea behind such a registration scheme is to simultaneously pre-configure and execute a bundle for a given engine. An example of an rS unit's double configuration register setup is shown in Fig. 3.5. As seen in the figure, while the multiplexer is set to execute one configuration register set (select bit), the other may concurrently be configured to effectively conceal any setup overhead.

Using double configuration registers, the proposed aggressive pipeline is provided in Fig. 3.6, consisting of two engines with a single-thread's workload mapped to each. As seen in the figure, such a pipeline now eliminates both conventional front-end limitations, in addition to configuration overhead. The scheduler however must now keep track of bundle dependencies, predictions, and completions for all running engines. A more illustrative example which explains the pipeline in further detail, referring to the execution process presented in Fig. 3.2.

As seen in Fig. 3.2 (Step 1), the scheduler outputs an ID i.e. configuration (config) address when a bundle is ready for execution. The bundle's config memory address is then used by each engine structure's memory bank to read in the configuration bits contiguously stored (Step 2a). These bits are then loaded onto each engine component's configuration register, where any external values (EARF

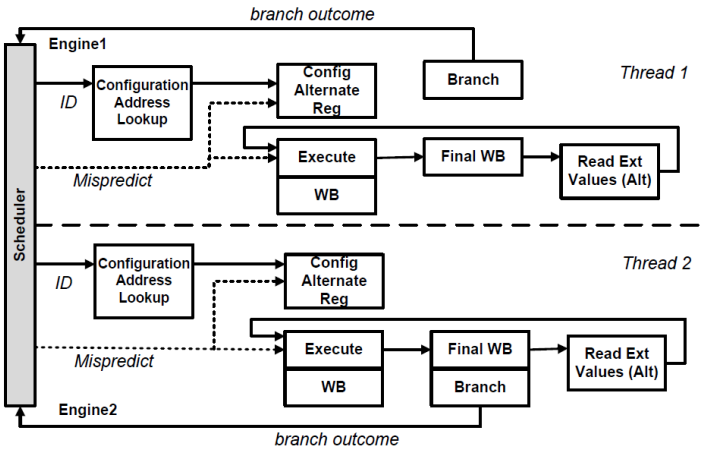


Figure 3.6: ConSSTEP Aggressive Pipeline (2 Engines)

reads and/or cache loads) needed are simultaneously obtained and stored to the **Read Register Buffer (RRB)** (Step 2b). This phase is referred to as the *engine configuration and external read* process (external values required for the bundle to commence execution). Once loaded, each individual bit in the configuration register is used to dictate the unit's control logic per clock cycle during execution.

Therefore once the configuration and external read phase have completed, execution commences (Step 3b). As instructions execute on the FUs, data is propagated throughout the rS interconnect as topologically configured, where data may also be temporarily stored in the distributed rS units if needed later by proceeding instruction(s). External FUs not present within the main engine may be shared between engines for more complex instruction execution such as load/store units (LSU) for memory references. Such unit interfacing is completed on behalf of an engine's RRB and **Write Register Buffer (WRB)**. Each engine also possesses a conditional branch unit for executing branches (as bundles terminate upon conditional branches) (Step 4a), which is also predicted by the scheduler's branch predictor. Therefore when the correct branch outcome is computed by the engine during execution, it is sent to the scheduler to verify its prediction and signal bundle completion. At this time, any final data values pending in the WRB are also written or stored to the external register file and cache, respectively. Since the engine has already been pre-configured for the next bundle, it may read any external values necessary while the scheduler verifies its branch prediction. Assuming a correct prediction, the new bundle then continues to execute where the next predicted bundle configures the alternate register set (i.e. the register set which has just completed

execution, assuming the correct branch prediction outcome). Further architectural and execution details, including branch mispredictions are discussed in Chapter 5.

3.6 Execution Example

An example of a bundle executed on a four-FU (4-FU) ConSSTEP engine is presented in Fig. 3.7. The bundle to be executed (represented as instructions here) is provided to the right of Fig. 3.7, where the circled numbers signify an instruction's execution time as determined by the PhysC. Instructions are displayed as *<operation, result, operand1, operand2>*. The engine's path of execution for the listed instructions is provided on the engine's datapath, temporally and spatially configured with the bits loaded to each component's configuration register. Various dashed lines and colours used in the figure represent execution in different clock cycles.

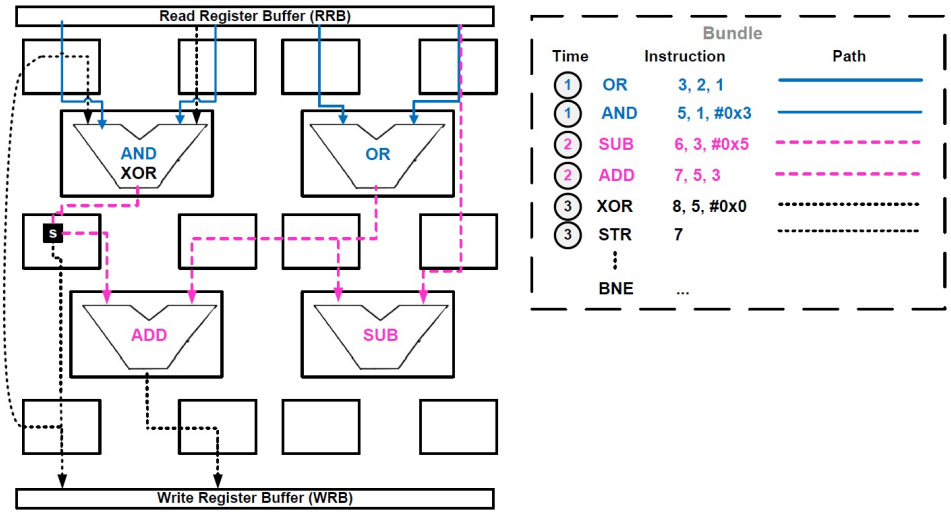


Figure 3.7: ConSSTEP Execution Example for 4-FU Engine

In the first cycle, the OR and AND instructions are executed (opcodes buffered in their respective FUs), each obtaining their source data from the RRB (i.e. external register reads and/or immediate values buffered during configuration). The result generated by the OR instruction is then needed twice thereafter by the SUB instruction executed during the second cycle. The first cycle's AND instruction is also a producer for two other instructions, once in the next cycle for the ADD instruction, and once in the third clock cycle for XOR. Since the operand ('5') is

required at two different times, the result must be both propagated and stored in the first clock cycle. Therefore during the 3rd clock cycle, the operand is obtained from the rS unit storage (marked 'S' in the 2nd row, 1st column rS), and propagated to its consumer as required. '5' is also a final register value and must be written to the external register file. In this example, the PhysC schedules the external register write during the third clock cycle, with the final store instruction (7) also written to the WRB and thereafter forwarded to the D-cache. As seen in this example, FUs may execute various instructions, one per cycle, where data flows throughout the rS interconnect. Instructions continue to execute similarly until bundle termination, dictated by the branch.

3.7 ConSSTEP vs Conventional CPUs

Now that the ConSSTEP process has been sufficiently reviewed, this section discusses the benefits of ConSSTEP in comparison to conventional processor structures, interconnects, and pipeline stages.

3.7.1 Execution

Standard CPUs (including VLIWs) consist of a static set of predefined FUs and/or sets of FUs (i.e. two ALUs, one Multiplier/Divider, one Branch etc) which dictate a core's issue width. CPU's execute an out-of-order, yet fairly sequential instruction stream creating a limited view of an application's data dependency graph using the IQ. The IQ monitors its dependencies through complex control logic and associative searches, dispatching instructions once ready. Due to this backend design, a restriction exists with respect to the type of instructions and maximum issue width a CPU may support, and is further limited within SMT models. This limitation brings forth the need to execute more demanding workloads on heterogeneous multi-cores to compromise between single and multi-threaded workloads. However only so much TLP and ILP may be extracted from these workloads as performance is still limited by sequential execution and a single core's architecture [38, 50].

ConSSTEP's back-end consists of execution engines, each which may comprise of a different number and combination of FUs that execute concurrently and independently. It is arguable however that not all FUs are needed at a given time,

considering both computationally- and memory-intensive workloads. On the contrary, the inclusion of such FUs and rS units allow engines to possess flexible issue widths and distributed storage locations. Such a backend design also *eliminates* effects of:

- Continuous access to a centralized register file, with increased read/write port requirements for wide issue execution
- Unscalable issue widths
- Fine-grained dispatch
- Tile-based hotspots (for distributed architectures)
- Unnecessary broadcasts to the IQ, bypass network, and register file
- Increased logic complexities and the general unscalability of multi-stage bypass networks.

Encompassing a variety of FUs and engine types also allows ConSSTEP to adapt from simple to more complex workloads on a single chip, in addition to varying degrees of TLP while extracting higher performance and ILP. It is also less expensive in terms of hardware, area, and power to include a few additional FUs and rS units versus several processors in the case of multi-cores, if similar or better performance is achieved.

3.7.2 Storage & Interconnect

Several works have addressed issues of value lifetime and redundant register file bandwidth [6, 8]. According to Tseng and Patt [6], 80% of these values have an average lifetime of 32 instructions or fewer. Using the PhysC to form bundles and extract instruction dependencies, experimental testing reveals that on average a bundle includes approximately 28 instructions, and does not exceed 80 instructions. Hence majority of operand lifetime may be maintained within a bundle through such coarse-grained execution (topic revisited in Chapter 9.4.2).

Optimistically assuming computation energy costs continue to decrease through voltage scaling, that frequency remains fairly constant, and increasing performance will remain a constant goal, the cost of data movement within a processor begins to dominate energy efficiency [1, 43]. It is therefore important that data movement be

restricted to achieve the increasing benefits of performance scaling. This signifies that data must be kept as local as possible, and that either the size of the local storage must be increased (preferably in a distributed manner to maintain access speeds) and/or producer-consumer dependencies between instructions must be better exploited.

rS interconnects within a ConSSTEP engine achieved interconnect scalability using a torus-like topological infrastructures which possesses temporal storage properties and single-cycle multi-hop data communication between dependent instructions. The rS units therefore maintain data locality and propagate multiple results to their consumers *only when required* while also eliminating the need for register file hierarchies (discussed in Chapter 4). Furthermore, configuration data generated by the PhysC eliminates the need to propagate tags throughout the network as the case of dataflow architectures, further mitigating data movement and improving energy efficiency.

3.7.3 ConSSTEP vs VLIWs

ConSSTEP shares similarities to the static compilation approach of VLIWs. VLIWs however general extend instruction sets with specific instructions, whereas ConSSTEP extends the functionality of the ISA by generating control logic for a configurable underlying architecture. By also using a coarse-grained approach, ConSSTEP provides more flexibility than VLIWs, extracting both operation parallelism and data transport patterns while maintaining software compatibility. ConSSTEP may also execute a variable amount of instructions per clock cycle per thread for multi-threaded workloads, whereas VLIWs execute single thread workload which cause core underutilization. The drawbacks of static compilation techniques such as cache misses, are also mitigated in ConSSTEP with sufficient instruction parallelism from other concurrently executing threads.

Although certain aspects of the PhysC may resemble a VLIW-like compiler, the PhysC adopts certain principles from co-designed VMs to support standard compilers, however implementing a reconfigurable (and/or adaptable) underlying architecture. VLIWs conversely employ customized primary compilers specific to their underlying architecture during compilation, causing certain software compatibility issues (i.e IA32/IA64, backwards compatibility issues, and the need for re-

compilation). Evidently, the notion of a PhysC allows applications to be distributed on a wide range of processor types to increase efficiency and promote the processor's role, while concurrently managing to conceal the software from the target architecture and avoid the need for re-compilation.

In terms of the architecture, VLIWs also require demanding amounts of register file ports for wide-issue which contribute to additional power consumption, logic complexities, and latencies. Conversely, ConSSTEP promotes both scalable logic and interconnect solutions, reducing register ports, accesses, and storage requirements significantly, while supporting scalable data propagation.

3.8 Tradeoffs of ConSSTEP

Although ConSSTEP provides many potential performance, area, and energy efficiency improvements over conventional processors and SMTs, there are trade-offs to such advantages: configuration memory requirements, and the hardware/software co-designed PhysC.

Generally, the advantages of configurablity are often at the expense of additional hardware logic, latencies, and memory storage requirements. These factors are especially true for architectures which are augmented with configurable logic, such as the previous work discussed in Chapter 4.4 (i.e. MorphCore, CoreFusion etc). Conversely ConSSTEP directly considers such constraints by completely redesigning the datapath to accommodate configurability in a simplistic manner while enhancing the single-thread performance of multi-threaded workloads. To mitigate latency overhead, ConSSTEP distributes configuration memory and supports an aggressive pipeline. ConSSTEP also integrates configurable hardware within its datapath structures to eliminate unnecessary hardware logic, and is able to increase operational frequency in comparison to conventional processors due to its simplistic logic. The aggregated memory storage requirements for the configuration data however are much higher in comparison to a typical processor's I-Cache (verified in Chapter 9.4.1).

ConSSTEP's extra memory storage requirements however are at the advantage of eliminating several front-end pipeline and OoO re-ordering structures, decreasing the dynamic power and area overhead of a processor. In such a case, the elim-

inated front-end (dynamic) logic is replaced with configuration caches which are used sparsely in comparison to every clock cycle as the case of superscalars (i.e. once per bundle). The configuration logic used to mitigate the processing overhead of conventional CPUs however is generated at the cost of a increased software complexity and hardware/software trade-off. That is, the software-based PhysC which generates configuration logic is used to eliminate hardware-based instruction processing overhead and potentially increase issue width.

As previously discussed, the works of Transmeta's Crusoe [44] and Nvidia's Denver [48] have attempted to implement runtime co-designed VMs (software) to eliminate front-end structures and accelerate performance on-chip at the trade-off of software complexity. The PhysC concept takes an alternate approach to the Transmeta and Nvidia translation/processor envision. Rather than converting one ISA instruction to another on-the-fly (where both Transmeta and Denver possess an in-order VLIW-like underlying architecture), the PhysC promotes a static and compatible macro-processing translation flow which generates control logic for configurable processor architectures, also mitigating several VLIW bottlenecks. Thus the PhysC compromises between the benefits of a *smarter* static compilation process in comparison to VLIWs while supporting the binary compatibility objective of co-designed VMs. The PhysC concept however requires much more research, especially regarding the elimination of various memory and aliasing issues. For this reason, the PhysC is implemented using a trace-based technique in this work to mimic steady-state behaviour and leave such issues as future work.

The concept of a PhysC therefore remains the greatest trade-off of the ConSSTEP architecture, and is heavily reliant on its feasibility. As history dictates however, VMs themselves were considered as mere academic curiosities which provided extraordinary system flexibility for certain unique applications [51]. The progress and research placed in such VMs within the past decade clearly demonstrate that programmers and research have greater enhanced the area of VMs, and therefore a PhysC concept may in fact be feasible within the near future.

Finally, it is worth mentioning that the static approach invoked by the PhysC is also at a loss of the pure dynamic execution style typical of superscalar OoO CPUs. As discussed in Chapter 2.8, conventional OoO execution allows a core to dynamically determine operand status and execute, versus a static compilation approach

which assumes a certain latency during instruction scheduling. Thus if an instruction takes longer than anticipated, i.e. a cache miss, the processor must stall as the compiler can not anticipate the miss. Conversely OoO superscalars may continue executing independent instructions to effectively increase performance in comparison to VLIW. However considering that ConSSTEP directly improves the performance of multi-threaded workloads and decreases the processing overhead per instruction, such disadvantages are relieved. The work of Hilly and Seznec has also demonstrated that even an in-order 4-thread SMT may reach 85% of the performance of a 4T OoO processor through multi-threaded workloads[52]. Accordingly, the effects of a static compilation approach are mitigated by supporting such multi-threaded workloads, OoO compiled execution, and a nuanced underlying architecture.

3.9 Summary

This chapter provided a general overview of the proposed ConSSTEP architecture, outlining the mechanics of the core. The chapter provided a brief outline of the compilation procedure, the micro-architectural and architectural flow, any any assumptions made throughout the process. An example illustrating the execution flow for the proposed core model was provided. Thereafter, the chapter outlined the advantages which the ConSSTEP architecture possesses over conventional processor methods, where the chapter concludes with a discussion on the trade-offs to which ConSSTEP achieves such advantages over conventional core – namely additional memory to support an re-configurable underlying architecture and the compromise between hardware and software complexity of the PhysC.

Accordingly, now that the ConSSTEP methodology is understood, the next chapter discusses related work of various processor system research, and compares each work to the ConSSTEP architecture. Further details pertaining to the ConSSTEP architecture and compilation process are provided thereafter in Chapter 5 and 6, respectively.

Chapter 4

Related Work

4.1 Introduction

This chapter provides background on several related processors which have attempted to deviate from conventional models. To discuss thesee works, this chapter is categorized into the following sections: hybrid data-flow architectures, distributed and coarse-grained models, reconfigurable architectures and CGRAs, and other architectural models. All models are then contrasted and compared to the method proposed by ConSSTEP.

Hybrid data-flow architectures attempt to expose data-flow like execution in conventional control-flow instruction streams to increase ILP and TLP. Data-flow machines previous to these hybrid models were successful, however experienced several software compatibility issues and required custom programming languages to eliminate control-flow instructions. The hybrid approach discussed in this chapter however maintains software compatibility by integrating ISA extensions and/or custom compilers to support data-flow techniques (i.e. data triggered execution) for a given architecture. Accordingly, the first section of this chapter discusses the eminent hybrid architectures of WaveScalar [30] and TRIPS [34, 53]

Distributed and/or Coarse-grained models aim to eliminate the centralized structures and/or fine-grained instruction execution of conventional CPUs, respectively. Accordingly, distributed models duplicate and simplify compute units, dispersing the structures throughout the core. Such units work concurrently to provide higher TLP and ILP in comparison to conventional cores. Likewise, coarse-grained models

employ custom compiling techniques to group instructions for coarse-grained execution, mitigating some of the control-flow limitations imposed on conventional processors. Accordingly, this section describes the eminent works of Task Super-Scalar [54], BRAID [6], WiDGET [55], and CRIB [50].

Next, reconfigurable architectures and Coarse-Grained Reconfigurable Accelerators (CGRAs) are discussed. CGRAs propose the idea of integrating configurable accelerators into a pipeline or as co-processor for the main CPU, such that the accelerator may dynamically configure for recurring phases of code. Since the use of dedicated accelerators may only benefit certain applications, the concept of reconfigurability poses as an attractive solution for a wide selection of workloads. Similarly, the reconfigurable architectures discussed in this chapter possess certain properties which allow a core to transition from one core to another dynamically (i.e. in-order SMT to OoO single-thread and vice versa), compromising between performance and energy efficiency. Works discussed in this section include DySER [56], MorphCore[10], and CoreFusion[57].

Finally, other processor models which do not fall into any of the previous categories with similar objectives to ConSSTEP are discussed in *Other Architectural Models*. The models discussed include Stream Processors, and Transport Triggered Architectures (TTAs).

4.2 Hybrid Data-flow Architectures

Many previous works have applied hybrid data-flow concepts to deviate from the conventional control-flow OoO processor models to exploit dataflow, instruction and data-level parallelism. Such architectures aim to provide both execution efficiency and compatibility with legacy software.

Wavescalar is a custom compiler which works on an extension of the Alpha ISA, designed for dataflow-like execution [30]. The compiler generates "waves" (i.e. bundles of instructions) which execute on the WaveCache processor. The WaveCache processor itself is a tile-based processor, where instructions execute on processing elements (PE), each consisting of a five-stage dynamic pipeline. The internal architecture is very much similar to a conventional CPU, however functioning on the data firing rules and tokens of data-flow models. The placement of instructions is deter-

mined both at compile-time and runtime, where a hierarchical interconnect is used to provide data-flow communication among dependent instructions, which may add latencies for the data bypassing of instructions that are not within the same vicinity.

Although Wavescalar is a novel approach for transitioning away from a conventional CPU model, start-up costs, dynamic instruction loading, discarding, and context switching exhibit large latency penalties due to the hierarchical interconnect and outer bound L2 data cache placement. Consequently, Wavescalar's performance is negatively affected for memory-intensive applications. Sequential workloads also suffer a performance loss when highly dependent instructions become scattered throughout the WaveCache, increasing communication costs and interconnect traffic. The compiler and WaveScalar ISA are also strictly dependent on the WaveCache architecture for instruction placement, and so other execution models do not necessarily benefit from the WaveScalar ISA.

The **TRIPS** model is a distributed, polymorphic architecture, capable of statically configuring its hardware to support variable instruction granularities and diverse workloads [34, 53]. TRIPS functions on the innovative and custom Explicit Data Graph Execution (EDGE) ISA for mixed dataflow/control-flow based execution. EDGE uses the notion of hyperblocks (i.e. blocks of instructions), where the role of the compiler is to statically schedule the blocks on the execution model and exploit dependencies critical to performance. One of EDGE's most defining features however is its direct communication among dependent instructions, achieved by embedding target operands within an instruction so that results produced can be directly forwarded to targets versus the register file for dataflow like execution. The TRIPS processor itself consists of five different types of tiles that are duplicated and distributed across a platform, where each tile provides a specific pipeline function, interconnected through various micro-networks.

Although TRIPS provides a significant effort to deviate from conventional models, the core is only able to obtain limited performance gain due to its microarchitectural organization. As certain micro-networks and tiles lie on critical paths between dependent instructions, operand delivery and/or data communication act as bottlenecks for purely sequential applications, contributing to unnecessary congestion. Memory and communication overhead also prove problematic for memory-intensive applications due to tile positioning of memory. Similarly, although there are many

advantages to the EDGE ISA, there are also limitations regarding block and branch prediction, the high fanout of instruction targets needed to deliver producer results to their consumers, and compiler dependencies on a grid-like architecture. Furthermore, embedding more information into instructions contributes to fairly large code sizes, storage requirements, and instruction cache misses. Since the TRIPS architecture employs a custom compiler, the EDGE compiler also lacks certain optimizations in comparison to other eminent compilers.

Comparison to ConSSTEP

Unlike TRIPS and WaveScalar, ConSSTEP may employ any ISA and/or compiler by utilizing a two layer compilation approach, eliminating burden on the primary compiler and the need to implement ISA extensions. It is true however that both TRIPS and WaveScalar may also apply a PhysC technique to their works. Therefore when directly comparing the TRIPS and WaveScalar architecture to ConSSTEP, both TRIPS and WaveScalar maintain a conventional n-stage pipeline and simply apply it in a spatially-oriented manner.

Specifically, in the case of TRIPS, tiles are scattered throughout the core, where each is responsible for a certain pipeline function. ConSSTEP completely deviates from such a pipeline, allowing the PhysC to handle the front-end stages, where the actual hardware is solely responsible for execution and data writeback to improve energy efficiency. Engines also maintain one transport network, whereas TRIPS requires six, contributing to various contention and hotspot points due to its large, mesh topology.

Conversely WaveScalar also spatially distributes execution engines within a core as the case of ConSSTEP. WaveScalar however uses a complete dataflow technique to transport data through a hierarchical interconnection per wave. Such a triggered dataflow method requires complex tag manipulation and data routing among FUs. Its hierarchical interconnect (similar to TRIPS) also incurs additional and undeterministic delays during data transport. Conversely, ConSSTEP maintains all operand communication within an engine with minimal contention due to the rS interconnect's temporary storage system, providing configurability and workload adaptation. ConSSTEP also maintains a conventional memory system layout, while amending the pipeline functionality to match or exceed the performance of conventional processors.

4.3 Distributed and Coarse-Grained Architectures

Task SuperScalar (TaskSs) [54] provides a well-designed solution for mitigating the performance effects of control-flow instruction streams by creating tasks using the OmpSs programming model. The OmpSs model is employed to support parallel task execution and is extremely similar to the OpenMP paradigm (and in fact has influenced several OpenMP v4.0 features [58]). OmpSs programs are compiled using the Mecurium compiler, and spawns a task-generating thread which sends compiled tasks to the TaskSs front-end. The front-end then decodes inter-task dependencies at runtime in a similar manner to conventional superscalars, however at a coarser granularity. Once a task's dependencies have been met, the task is dispatched to the backend which consists of a series of interconnected superscalar processors to execute the ready task(s).

Although TaskSs is able to mitigate several issues of coarse-granular execution by employing an amended front-end and OmpSs programming model, the TaskSs back-end simply consists of multiple processors. Therefore the tasks must traverse through another processor pipeline to execute its intra-task instructions. Hence area overhead and power consumption are not efficient nor ideal.

The **BRAID** [6] architecture aims to provide a wide execution core with OoO performance and in-order complexity by exploiting the small fanout and short lifetimes of values produced by a program [6]. Braids are bundles of instructions generated using an augmented compiler, ISA, and microarchitecture. The BRAID compiler invokes a binary profiling and translation process to extract producer-consumer dependencies for each instruction, embedding hints within the instructions for the BRAID architecture to exploit at runtime. The architecture itself consists of reduced complexity OoO pipeline, where FUs are replaced with Braid Execution Units (BEU). BEUs execute braids in-order and bypass values to a BEU's issue queue, internal register file (for short-lived operands) or the external register file (for other BRAIDs and BEUs to use), with the aid of the ISA extension. Such a design is able to reduce each structure's complexity and the size of the register files.

Significant energy savings come from BRAID's shorter rename stage and reduced

register file accesses. BRAID is also able to give a slight performance increase (1.5%) when compared to an OoO core equivalent.

The **Wisconsin Decoupled Grid Execution Tiles (WiDGET)** [55] architecture aims to provide an energy efficient computing structure for OoO single-thread performance using an in-order approach, decoupling thread management using several distributed and clustered simple in-order Execution Units (EUs). Specifically, the EU's may execute one instruction at a time (either int ALU, FP ALU, or Address Generation Unit), where several EUs work concurrently. Once executed, instruction dependencies are redirected to their dependent EU through instruction steering logic among the EU clusters. WiDGET aims to provide such scalability without the need for compiler and/or ISA support. Accordingly, the architecture supports the SPARC ISA while implementing a priori static allocation policy to map instructions to EUs and provide performance efficiency.

Experimental testing proved that the in-order resources which WiDGET invokes are able to provide OoO-like aggressive single-thread performance for majority of computationally-intensive SPEC benchmarks. However great performance losses were observed in memory-intensive applications due to its architectural layout.

In attempts to depart from the confinement of traditional OoO processors, the **Consolidated Rename Issue Bypass (CRIB)** [50] architecture implements an innovative approach to the rename, issue, and bypass pipeline stages through structure consolidation while also maintaining compatibility with current software stacks. CRIB eliminates large multi-ported physical register files, reservation stations, Register Alias Tables (RAT) and the ROB to provide performance and energy efficiency through in-place execution. Specifically, the CRIB architecture employs a standard gcc compiler and the x86-64 ISA, where instructions are grouped into bundles of four decoded micro-ops during runtime according to the fetched instruction stream. The CRIB architecture consists of many in-place execution engines i.e. *entries*, where multiple entries form a *partition*. An entry's architecture is arranged in two-dimensions, where data is latched horizontally upon completion, and computed/propagated vertically upwards (from bottom to top) in program order. Operands are obtained from the bottom row's Architectural Register File (ARF) columns and routed vertically upwards to and from appropriate source and destination columns for ALU row computation. Once all four (micro) instructions have

been executed in every entry and partition, they are committed in a circular fashion by all CRIB partitions, adhering to program order and eliminating the need for the ROB.

CRIB relies on a program ordered instruction stream to determine its instruction chains. Therefore instructions grouped together do not necessarily comprise of producer and consumer dependencies, limiting the possibility to execute longer instruction chains and utilize partitions more efficiently. Consequently such PC-based ordering imposes negative effects on CRIB performance.

Comparison to ConSSTEP

Both WiDGET and BRAID attempt to provide the benefits of OoO execution with in-order complexities and energy efficiency. In the case of BRAID, compiler and ISA augmentations are required by the core which cause certain compatibility issues for general purpose computing. Conversely, WiDGET is able to provide full compatibility, however its general layout and the use of small cores operating in lock step contribute to inter-core communication costs which prevent the architecture from attaining further performance. With regards to value lifetime in the case of BRAID, having multiple copies of architectural register files and an external register file does not completely address nor provide a complete solution to the data transport problem.

Conversely, ConSSTEP addresses compatibility issues and value lifetime by allowing the secondary compiler to relieve the burden of redesigning the primary compiler, while also extracting instruction dependencies within bundles. No amendments are needed by the ISA or primary compiler, while an rS interconnect is used for both communication, bypassing, and temporary storage between dependent instructions to completely mitigate unnecessary data movement and/or register file hierarchies.

With regards to the TaskSs architecture, although the work is successful at addressing coarse-granular dependencies dynamically through hardware, it does not directly address the back-end execution bottleneck. In fact, TaskSs increases power consumption while requiring the OmpSs programming model which may require reprogramming and re-compilation for certain applications.

Finally, when compared to CRIB, the CRIB core does not extract instruction dependencies in the same manner as a PhysC, contributing to additional dependencies and latencies. Each CRIB entry supports 4-entries in a static-like nature, while consolidating many pipeline stages to decrease on-chip power consumption. The benefits of a static dataflow-like execution which CRIB tries to extract are fundamentally limited by the architecture, instruction streams, and the need to maintain compatibility with the x86 ISA. On the contrary, a ConSSTEP engine may comfortably support and execute a multitude of instructions per engine in a dataflow-like manner while also avoiding the burden of ROBs, register renaming, physical register files, and the issue stage. ConSSTEP avoids ISA and compiler compatibility issues using the PhysC, and is able to mitigate the power consumption of many other pipeline stages using a completely nuanced approach to a processor's architecture.

4.4 Reconfigurable Architectures and CGRAs

The reconfigurable style of computing has received much attention in the past few years as it has the potential to provide both higher performance and system flexibility at the expense of certain programming effort. Intel's recent acquisition of Altera has also steered commercial research towards the possibility of reconfigurable computing, which would especially be beneficial for high performance applications which require application acceleration.

An FPGA's bit-level configurable approach however requires very fine-grained application customization, a significant increase in design effort, and long compilation times which prove problematic for general purpose computing. Coarse-Grained Reconfigurable Accelerators (CGRA) have been proposed to mitigate FPGA effects, raising configurability to the word-level similar to ConSSTEP and reducing the amount of configuration information necessary for the dynamic customization of computing systems. CGRAs act as accelerators and/or co-processors to monolithic CPUs, and encompass several internal computing elements which are interconnected for dataflow-like execution [56, 59]. CGRAs increase performance however at the expense of ISA, compiler, and microarchitecture modifications to support custom instructions which redirect applicable code phases to the backend CGRA unit(s). Certain CGRAs also require custom programming languages, design flows, and OS support however [59–61]. Below describes examples of CGRAs also briefly covers eminent works in the area reconfigurable computing architectures.

DySER (Dynamic Specialized Execution Resource) [56] is a CGRA embedded in the backend of a monolithic processor. DySER requires a LLVM designed compiler and amends the SPARC ISA with custom extensions for interfacing DySER to a conventional processor. The compiler is responsible for profiling and evaluating application kernels, determining if sufficient acceleration exists to benefit from a dedicated pipeline accelerator. The compiler then selects the most prominent recurrent kernels, generating memory and computation data for DySER to reconfigure dynamically according to the kernel's dataflow requirements. Therefore, when an accelerated phase of the program is approaching, DySER dynamically reconfigures to the kernel's data-flow, acting as a dedicated accelerator within the pipeline.

Although DySER displays significant energy and performance savings, the data and control communication overhead penalty between the accelerator block and the CPU pipeline is high. A heavy reliance also exists on the CPU pipeline to provide DySER with data. DySER's integration into the CPU pipeline therefore complicates an already fairly complex front-end, where DySER portrays intrusive integration for more high performance processors. Furthermore, many compiler amendments are required to embed and provide phase predictions in advance to dynamically configure DySER and conceal latency overhead. This is especially difficult and problematic when considering irregular control-flows and speculative execution. Finally, applications that do not possess sufficient recurring phases do not benefit from DySER, in turn increasing static power dissipation.

Other tightly-coupled CGRAs include **Matrix** [60] and **CHIMAERA** [62] etc. which are also integrated within monolithic backend pipelines for executing data-flow-like phases of an application. Similar to DySER, the ISA, compiler and processor microarchitecture require modification to support custom instructions which redirect applicable code phases to the CGRA unit. Conversely, loosely-coupled CGRAs such as **PipeRench** [61] and **FPCA** [59] etc., act as coprocessors to the CPU and thus are located externally from the main core. Since the CGRA is not within the processor pipeline, such coprocessors may be customized for larger application phases while achieving high energy efficiency for a given kernel. This flexibility however often comes at the expense of custom programming languages, design flows, and compilers which contribute to software compatibility issues and an increase in offload/communication latencies for general purpose processors.

MorphCore [10] is a processor core which has the ability to run as a highly-threaded in-order SMT core for multi-threaded workloads, or an aggressive OoO core for single (active) thread workloads. MorphCore takes advantage of the fact that aggressive OoO cores are ideal for providing high single-thread performance, whereas SMT workloads may achieve almost the same performance as an OoO core using in-order cores at a fraction of the power. In order to achieve such a compromise, the architecture monitors the number of active threads in a workload, reconfiguring from one core to the other when a certain "thread" threshold is reached. Specifically, when the threshold is surpassed, the core runs as an in-order SMT, whereas when only a single-thread is active, the core runs as an aggressive OoO engine. Mode switching is at the cost of draining the pipeline, spilling and refilling the architectural registers with the active thread(s) data, and shutting down the required pipeline structures when switching to in-order SMT mode. Consequently, depending on the overhead of the workload, such switches may be fairly expensive.

Due to MorphCore's reconfigurable properties, the core can not run at the same frequencies as its core equivalent, i.e. overheads from switching and the additional pipeline logic required to reconfigure the structures add to critical path latencies. Therefore the core can only achieve performance in close range to the throughput optimized cores, however still experiencing the thread blocking and sharing issues of a conventional CPU. MorphCore however is able to provide improvements in energy efficiency.

CoreFusion [57] consists of multiple and identical two-issue OoO cores, where a bus is used to connect the sets and respective L1 I- and D-Caches to provide coherence (all cores sharing the L2 cache). When required, each core may execute independently for high thread-level performance, or be fused in groups of 2-4 cores to form larger more aggressive cores for wide-issue execution and efficient single-thread performance. Additional hardware logic is necessary to coordinate the cores, however contributing to additional energy requirements and limited performance. Furthermore, larger core configuration have lower performance and higher energy consumption in comparison to a conventional OoO due to the latencies incurred in mode/content switching, also requiring pipeline flushes and data migration as in the case of MorphCore. Experimental testing reveals that MorphCore is able to exceed the performance and energy efficiency of CoreFusion however.

The **ReMap** [63] and **ReKonf** [64] architectures also address reconfigurability in chip multi-core processor systems in alternate ways. Specifically, the ReMap architecture provides a solution for reconfigurable fabrics on heterogeneous multi-core systems, where the fabric is able to build and collapse on-chip communication buses for customized data transfers between cores according to an application's requirements. Therefore ReMap provides more flexibility in comparison to hard-wired busses to overcome limitations of contention in general purpose multi-processors. On the other hand, the ReKonf architecture reconfigures entire multi-core processor system environments, activating a certain multi-core setup (i.e. x-CPUs, y-caches) depending on an application's requirements, monitored dynamically by the system which detects such thresholds of operation.

Comparison to ConSSTEP

The processors and CGRAs presented in this section reconfigure to match and/or accelerate performance, with impositions that come at the hardware and latency cost of reconfigurability. Conversely, ConSSTEP provides reconfigurable properties without such overhead and performance losses exhibited in majority of these works, in fact ConSSTEP is able to achieve greater operational frequencies than conventional cores (see Experimental Results). Specifically, architectures such as MorphCore and CoreFusion require complete pipeline flushes and context saves in order to transition to different modes which ultimately contribute to performance loss. ConSSTEP on the other hand is able to conceal the effect of configuration overhead using a double configuration register approach, while only suffering configuration overhead during branch mispredictions and/or exceptional events. In addition, although MorphCore, CoreFusion, ReKonf and ReMap are able to reconfigure to different core architectures, the architectures still face the same impositions of conventional processor cores. ConSSTEP instead applies the concept of reconfigurability to directly address impositions of a conventional processor with a diverse design for increased ILP and TLP.

In comparison to DySER and other CGRAs, ConSSTEP is able to extract dataflow from kernels, but also from the general instruction stream, while integrating reconfigurable properties within the datapath itself. ConSSTEP conceals setup latencies and interacts with the memory in the same manner as conventional CPUs to provide performance gains and the energy efficiency sought by CGRAs. Fur-

thermore, the core mitigates the need for any additional logic complexities, while avoiding alterations to the compiler and ISA, with full software support.

4.5 Other Architecture Models

4.5.1 Stream Processors

Stream processors [65, 66] rely on an abundance of application parallelism and are optimized for the *stream/throughput-oriented execution* model. SPs aim to extract multiple levels of data locality, and the predictability of data accesses in order to accelerate the throughput-oriented application domain. Such processors contain a large pool of ALUs, and a two-level register file hierarchy to expose the deep storage properties and data locality typical of throughput-oriented applications. In order to reduce hardware complexities for such a register file organization, SPs invoke a VLIW-like underlying architecture, redirecting result operands to their appropriate destinations and register file levels.

ConSSTEP's goal conversely is to improve upon both latency- and throughput-oriented applications by limiting the disadvantages of SMT CPUs. ConSSTEP does not require an abundance of ALUs, and maintains compatibility with conventional compilers and ISAs. SP's ILP and fine-grained thread limitations are mitigated by invoking a PhysC using scalable engine structures and the rS interconnect which relieves the need for register file hierarchies. The PhysC also eliminates the need to use a static VLIW ISA which contributes to further software compatibility issues.

4.5.2 Transport Triggered Architectures

Transport Triggered Architectures (TTA) [8, 67] are a superclass of traditional VLIW cores. As opposed to solely extracting operation parallelism as in the case of VLIWs, a TTA-based compiler also exploits parallelism at the data transport level. Similar to the objective of BRAID, TTAs exploit the fact that value lifetime and register file bandwidth are largely redundant. TTAs therefore opt to completely redesign the datapath to provide data transport and issue-width scalability.

TTAs consist of several FUs and a transport network consisting of various transport buses. Each FU in the TTA is connected to the transport network using one or more of the bus sockets. FUs are triggered to execute when appropriate data

has been moved to its FU-triggered registers (similar to dataflow tokens), where the result is stored to a FU-result register. Operands may be obtained from FU-operand registers, FU-result registers, or the general purpose register file. Thus the compiler must maintain these data locality details while scheduling instructions. In certain cases, the transport layer for a single instruction may require several moves to/from the various register types (i.e. result register to operand register, etc). Consequently, for certain applications TTAs require high register to register movement, where the compiler must integrate optimization techniques so that they may avoid dead instructions and common operands to decrease register movement.

The compiler compatibility of TTAs raise issues for general purpose computing for reasons similar to VLIWs. Hence most realizations of TTAs have been employed for specific applications [67, 68]. The compiler design is also fairly complex, requiring customization depending on the transport buses, clustered connections etc. Conversely, ConSSTEP aims to exploit both operation parallelism and data transport for general purpose applications while considering engine compatibility. The rS interconnect mitigates the TTA's abundance of move instructions using the configuration data generated by the PhysC, where data is only moved if required by another instruction. Therefore ConSSTEP presents a scalable interconnect and feasible solution for exploiting value lifetime in general purpose applications, as opposed to TTAs which may only execute a fixed set of applications and must manually optimized customized interconnects according to an application's communication characteristics.

4.6 Summary

Several works in unconventional processor models were discussed in this chapter, comparing and contrasting the methods used to the proposed approach invoked by ConSSTEP. All models displayed various advantages, yet certain disadvantages which ConSSTEP addresses. Accordingly, this thesis now introduces details of the ConSSTEP architecture in the next chapter, with two-level compilation specifications discussed in the following chapter.

Chapter 5

Architecture

5.1 Introduction

This chapter outlines the architectural and microarchitectural details of ConSSTEP. Each structure in the architecture is thoroughly described in this section, including the rS units, Read Register Buffer (RRB), Write Register Buffer (WRB), Functional Units (FUs), the external architectural register file (EARF), and configuration process for concealing overhead. Floating Point (FP) execution is also discussed, in addition to exception handling, and data memory accesses. This chapter is presented prior to the compilation process so details of the ConSSTEP architecture and microarchitecture are well understood prior to discussing the compilation process.

5.2 rS Interconnect

The two types of rS units present in an engine's interconnect are referred to as external and internal. Referring to Fig. 3.2, the external rS are found at the very top of the interconnect, connected directly to the RRB and consist of four ports - two input and two output. An example of an external rS architecture's port design is provided in Figure 5.1. These rS act as an interface between the RRB and FU inputs, and connect the bottom rS units to the top of the torus topology. External switches do not require storage properties since they are directly connected to the RRB (which buffer data) and do not receive any FU output. Horizontal propagation is also not supported by external switches due to an xy routing protocol for the data transport of operands. Therefore data must first be propagated horizontally

in the lower levels of the torus network, and traversed upwards. These factors allow external switches to possess low hardware area and a reduced number of IO ports.

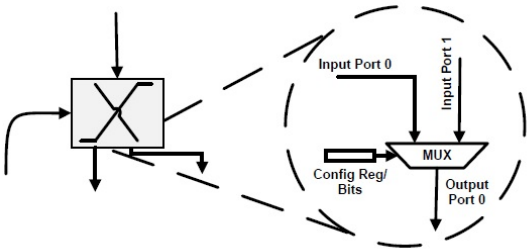


Figure 5.1: External rS Architecture

Internal rS are present in the main communication network (i.e. 2nd rS row onwards of Fig. 3.2) and consist of six ports- three inputs and three outputs, with an example of an output port shown in Fig. 5.2. As seen in the figure, all internal rS input ports consist of a 2:1 multiplexer that may 1) propagate input data, 2) output a stored value, or 3) simultaneously store and propagate a value. As shown in the figure, an output port multiplexes all input ports, and therefore all rS output ports may propagate data simultaneously to neighbouring rS, where all control bits are determined by the PhysC.

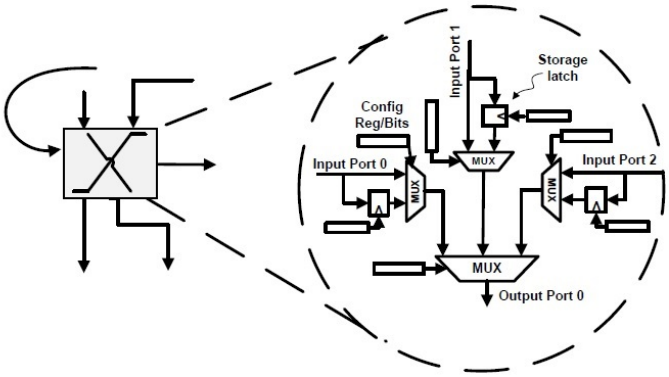


Figure 5.2: Internal rS Architecture

The rS interconnect was designed to support single cycle multi-hop traversal, signifying that any result may be sent to any FU source input (including its own) within one clock cycle. To support this feature, the frequency of operation is limited by an engine's critical path. As each ConSSTEP engine may vary in topological size, its respective maximum frequency also varies. As will be further discussed in

Chapter 7.3.2 however, *wire delays* only slightly increase as FUs are added to an engine. This is possible due to the torus topology which the rS interconnect adopts, stemming from the principles of Network-on-Chip (NoC) architectures.

The objective of NoCs are to replace lengthy buses in computing systems with short links and switches/routers to provide direct data communication between producers and consumers, especially beneficial for latency critical applications. Specifically, the switches are responsible for directing data throughout the network either statically (according to a packet's embedded header data), or dynamically (through Look-Up-Tables and algorithms in the switch). Conversely, the rS units invoke much simpler configurable logic which dictates the interconnect topology per clock cycle, versus the static or dynamic overhead which NoCs incur. The link length between the switches for both NoCs and ConSSTEP are short, and so their delays remain constant. Such links also allow for lower power and faster performance due to reduced contention per transfer, decreased wirelength, and lower bus traffic when directly compared to conventional bus networks [69].

By using a torus topology, performance of the rS interconnect is further enhanced by allowing opposite edges of the interconnect to be directly connected, effectively reducing contention and the number of rS units travelled between data dependencies. Such an advantage however comes at the complexity and cost of physically implementing two different length wires [69] i.e. managing the RC delays of the main short links and the longer outer grid links. However since the engine size is restricted to provide single-cycle interconnect latency, these disadvantages do not affect ConSSTEP as they are directly considered within the design. Performance is also enhanced with rS units as data travels in the worst case at a critical path length within one clock cycle and as verified in Chapter 7.3.1, is in the magnitude of picoseconds. On the contrary, conventional NoCs must make routing decisions every clock cycle and require pipelining per switch which takes several cycles to deliver data in comparison to a rS interconnect. Thus the rS design, its single-cycle delay, and configurabile topology allow for enhanced performance which is especially important in the latency-sensitive operations required in processor design.

5.3 Read Register Buffer (RRB)

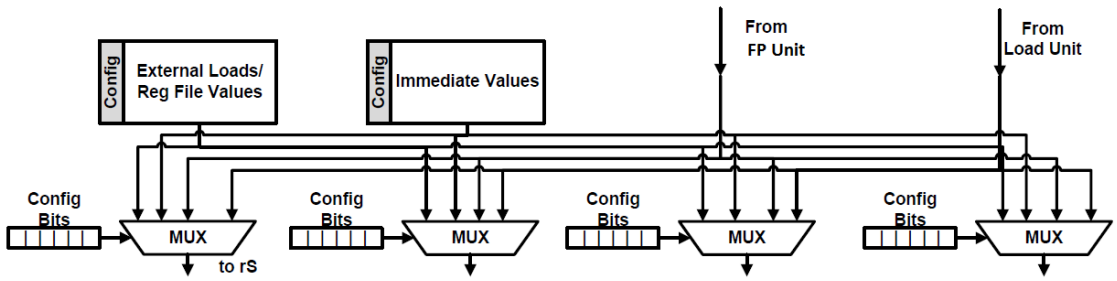


Figure 5.3: Internal RRB Architecture

The Read Register Buffer (RRB) is responsible for managing external data input and communicating such data to the engine's components as needed during execution. An example of a RRB's architecture is provided in Fig. 5.3. In general, RRB's manage:

- External register file reads and loads during configuration and communicate these values to the required rS during execution.
- Internal loads (requested by the WRB during execution) and communicate these loaded values to the required rS
- The communication of immediate values to the required rS, as configured by the scheduler/bundle data
- Incoming external unit results, which are communicated as programmed to the correct rS

Similar to the rS interconnect, the RRB is also provided with configuration data so that multiplexers directly connected inside the RRB to the rS interconnect may communicate data to each of the four external rS engine inputs as propagated.

5.4 Write Register Buffer (WRB)

The WRB's main objectives are to manage external instructions issued by the engine, and to communicate the data to their respective external units. Although the

WRB is not as complicated as the RRB design, it must properly interface external units and manage concurrent requests made by the engine. For this reason, a queue is included in the WRB to manage pending requests, but is infrequently required due to the scheduling techniques provided by the PhysC. The WRB must also handle the request and grant signals to external units shared between multiple engines. Based on these signals, if the RRB has not received a result after a specific time (i.e. four clock cycles for a load etc), the engine will stall until the pending data is received. Finally, the WRB manages the writeback of its final bundle values to the external register file as executed and configured.

5.5 Functional Units

FUs present within a ConSSTEP engine may be of any type including (but not limited to) ALUs (with barrel shifters), complex integer (including multiplier/divider), Multiplier-Accumulator units (MAC), and accordingly referred to as FUs. A low complexity conditional branch unit is also included to execute each bundle's final branch instruction. To support conditional branch execution, an engine contains a flag status register which is updated as instructions execute.

An example of the general FU architecture used by ConSSTEP is presented in Fig. 5.4. The configuration setup is similar to the rS, however the functionality of each FU/ALU is not programmable. Rather, the functionality of each FU is determine a priori, however the operation which the FU executes per cycle is determined by the PhysC. Once executed, the result is latched and thereafter propagated and/or stored as required in the next clock cycle. In the case of FP FUs or other more complex operations, such units and the respective data must be pipelined; however the overall architecture remains the same.

External to each engine are dedicated load/store unit(s) (LSU). This unit receives load requests made by the engine's WRB, fetches the cache data, and outputs the received data to the RRB, in turn propagating the value to the requested FU/rS. A load therefore takes approximately four clock cycles to execute, as standard in majority of conventional processors. The LSU also executes and sends stores to cache, where the Load/Store Queue (LSQ) and Miss Status Holding Register (MSHR) are responsible for handling any misses and memory dependencies.

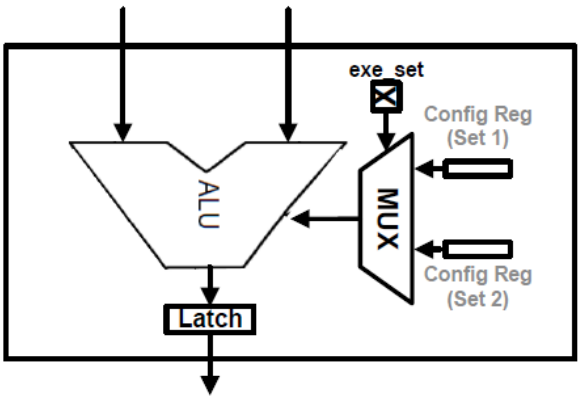


Figure 5.4: Internal FU Architecture (ALU example)

Any external units shared between engines are interfaced in a similar manner to the LSU. Such external units may include complex integer, multiplication and division units etc depending on the architecture and frequency of such operations in general purpose applications. A separate engine unit is also dedicated for FP instruction execution, where FP and integer data are forwarded to their complementary engines through the RRB and WRB (further details of FP execution provided in Section 5.7.2). Accordingly, the PhysC relies on an exposed pipeline to plan for the execution and transport of various instruction types and operations.

5.6 External Register File

A small architectural register file is placed in the ConSSTEP backend per engine, referred to as the external architectural register file (EARF). As mentioned previously, the PhysC is responsible for renaming and hence a large physical RF and its associated renaming logic is not required by the processor. The external RF is therefore an architectural RF with the basic registers required by the ISA. The EARF is accessed by an engine prior to bundle execution for reading bundle input values, used during execution, where final register values are written back to the EARF as they are produced within the engine for subsequent bundles to read as input data. Since ConSSTEP directly addresses the single-thread performance of multi-threaded processors, external register files are replicated i.e. one EARF per engine (per thread context), where data is shared between threads using shared L1 cache and lower level memory hierarchies.

5.7 Configuration & Setup

Each engine possesses its own configuration memory which is banked into several partitions to quickly configure engine components as shown in Figure 3.2. Specifically, separate memories are provided for the RRB, WRB, per row of ALUs, and two dedicated banks for every row of rS to allow for a maximum configuration time of 10 clock cycles (c.c) per engine (64-bit datawidth). The external register read and load configuration data also possess individual memory banks to obtain their values. For more demanding workloads, a multi-level cache system may be invoked by the configuration memory. In this work however, we assume the configuration memory footprint resides in the first level cache, where multi-level caches remain future work. Experimental testing displays that the average external register reads and writes required per bundle are approximately 4.8 and 3.4 respectively (considering a dual-port register file per engine), approximately 2 reads per FP engine, and an average of one to maximum two loads per engine. Hence such requirements meet the 10c.c/engine configuration time, where instructions exceeding such specifications are mapped to the next bundle (pending dependency analysis).

5.7.1 Setup Mitigation Techniques

As setup latencies may cause a significant amount of overhead in a reconfigurable architecture, two propositions for the ConSSTEP architecture are investigated in this work to reduce setup overhead - the 1) software and 2) hardware approach. Note however that the hardware approach is considered during the architectural explanations provided in the previous chapters and subsequent chapters.

Software

In the software approach, a variable configuration time technique is employed by the PhysC to reduce setup overhead. As applications and thread workloads vary, the average configuration data required for each bundles execution also varies. Consequently not all applications and execution engines require the same number of clock cycles during the reconfiguration process (i.e. less than or equal to 10c.c). Therefore, the concept of variable reconfiguration allows the PhysC to monitor an application's configuration time required per engine during compilation.

To support such a feature, the PhysC must monitor the maximum configuration

time required by each engine's component for all application bundles (per thread) during compilation. Thus using this technique, the *PhysC's Config Data Generation* (see Chapter 6) step is adjusted to keep count of the maximum configuration time required for all engine components. The PhysC then generates and sends 4-bits, i.e. representing x cycles, to each engine (once per application thread), dictating the absolute max configuration time necessary from the configuration memory address for x clock cycles (versus 10 static clock cycles per banked memory). Accordingly, an engine must possess a 4-bit register in this case.

Hardware

To provide ConSSTEP with a performance gain and decrease configuration overhead during execution and/or branch prediction, each engine unit in the hardware approach contains two configuration registers, for all components, shown again in Figure 5.5 - one set which dictates control for the currently executing bundle, and a second set of configuration registers which are configured concurrently for the next (predicted) bundle. Since configuration memory is distributed and each engine component contains its own dedicated configuration memory, the ability to concurrently execute and configure the engine for the next bundle is feasible. Thus, once the current bundle has finished executing, the next predicted bundle has already been configured and may read any external data values, executing while the branch outcome is sent and verified by the scheduler's predictor.

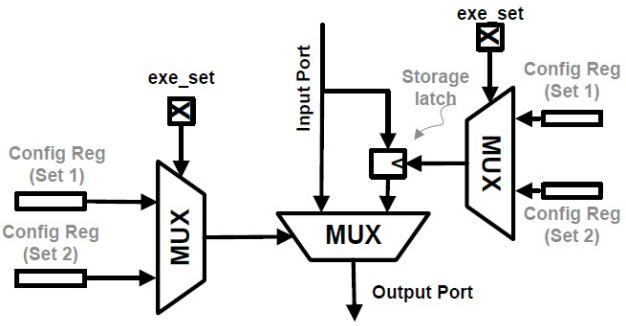


Figure 5.5: rS Unit - Double Configuration Register Setup

If the bundle is correctly predicted, the pre-configured bundle continues to execute while the next predicted bundle is sent to the engine by the scheduler, sufficiently concealing configuration overhead. If the prediction is incorrect however, the engine halts its execution (or memory read setup) and reconfigures for the correct

bundle while updating the predictor's branch history. Therefore ConSSTEP suffers a penalty on branch mispredictions only. Performance is also greatly enhanced in multi-threaded workloads as several engines execute concurrently in ConSSTEP, where each engine configures and executes in a concealed manner.

5.7.2 Floating Point (FP) Execution

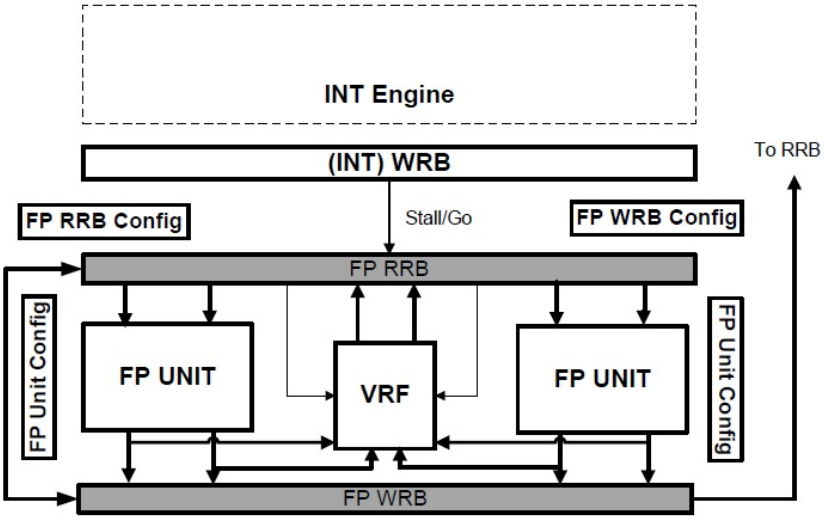


Figure 5.6: FP Engine Architecture

A FP engine is placed adjacent to its integer (INT) engine equivalent as seen in Fig. 5.6, where the integer unit communicates values through its WRB to the FP RRB. Considering the frequency of FP instructions, its expensive hardware implementation, and the fact that FP FUs are pipelined, invoking a 2-FU FP engine was determine to be the favourable solution by compromising between energy, performance, and PhysC complexity. To keep the FP engine simple, rS units are also eliminated, where the WRB is responsible for controlling dataflow, and the values temporarily held in the FP register file. Accordingly, the FP WRB is responsible for forwarding data to the RRB, FP register file, and/or back to the integer RRB depending on the instruction dependencies and configuration data provided. In the case where the application and/or thread assigned to the engine does not require FP support, the FP engine is disabled, determined by the PhysC.

The INT to FP engine dataflow exploits the concept of an exposed pipeline to route its data and execute operations. Hence, the pipeline latencies and underlying

architectures of the FP FUs must be known by the PhysC to form and map bundle dataflows. FP FUs therefore possess the same pipeline latencies across all engines, while supporting the same functions as the FUs of a conventional CPU.

5.7.3 Branch Prediction & Loop Acceleration

As previously discussed, the concept of branch prediction in ConSSTEP allows engines to start the next bundle's configuration process prior to explicitly resolving the current bundle's branch. Therefore, by implementing two configuration registers per component, ConSSTEP is able to mitigate setup overhead and increase system performance. Referring to Figure 3.2 (Step 4a) however, when an incorrect prediction is encountered, engine configuration must halt while the correct bundle ID is sent by the scheduler for engine reconfiguration. Since an engine contains two configuration register sets per component, it is possible however that misprediction penalties may be further reduced by consulting the alternate register set (i.e. the register set which has just completed execution).

As previously discussed, superscalar OoO processors may achieve loop acceleration by using dedicated pipeline units to monitor loops. These units may be duplicated per thread and/or shared by a SMT models. Conversely, processors with exposed pipelines, such as VLIWs and EPIC, use software pipelining techniques to detect loops statically in the code, where all dynamic and static models use loop unrolling compiler techniques to speedup execution. As opposed to invoking dedicated loop units or compiler techniques, ConSSTEP relies on its engine structures and branch predictor for loop acceleration. Specifically, as branch predictors may also detect the presence of loops within an instruction stream, the correct bundle may also possibly to be present in the alternate set of configuration registers.

The latency incurred for a correct branch prediction in ConSSTEP is simply reading register values from the external register file and commencing bundle execution. However in the cold case of loop detection (especially considering a bimodal predictor), the predictor may likely make an incorrect prediction. To mitigate such cold cases, upon a misprediction the scheduler also compares the bundle ID which has just completed execution to the ID of the correct bundle. In the case of a match, the misprediction penalty is mitigated at the expense of two additional clock cycles which allows the scheduler to send a control code to the engine (i.e. exe_set

bit shown in Figure 5.5) so that it may retain the last bundle and read new input register values. Thereafter, the branch history is corrected and the next bundle prediction is sent to the alternate set of configuration registers for configuration.

In the case that the previously executed bundle and predicted bundle *are both incorrect*, the engine configuration registers are cleared and reconfigured, enduring a 10 c.c penalty, while other threads concurrently execute in their respective assigned engines. Consequently in the case of smaller bundles, it is possible that values may have been written to the external register file during execution of the mispredicted bundle. In this case, a flash copy of the last committed bundle's register file is maintained. Accordingly upon a misprediction, the flash copy EARF becomes the active external register file while the other copy is restored as the engine is reconfigured for the new bundle. This latency penalty however is minor when considering the flushing and refilling of a conventional pipeline, and the general performance gains attained by ConSSTEP.

Finally, ConSSTEP also avoids the need for a ROB by dedicating one engine per thread, where the PhysC ensures instruction dependency requirements and memory ordering, stalling in the case of an unexpected event (i.e. cache miss etc). Accordingly once a bundle's branch outcome is verified as correct, the next bundle's instructions are also guaranteed to be correct, avoiding the need to maintain explicit fine-grained instruction program order through hardware structures.

5.8 Exception Handling

As discussed in Chapter 2.4.2, exception handling in conventional processors requires the pipeline to save state prior to the exception, flush subsequent instructions from the pipeline, execute the handler, and to restore the saved state. Since ConSSTEP takes an alternate approach to a processor's architecture which must also handle exceptions, bundles must also be created and distributed for exception handlers. Accordingly, when an exception is raised in the ConSSTEP architecture, the scheduler and offending engine interact, where a control code and "handler bundle" ID is sent by the scheduler. An engine however in this case "pauses" execution at the given offending instruction cycle. If the exception was due to a Translation Lookaside Buffer (TLB) miss or a similar case where the fault is generated external to the engine, the engine which raised or issued the exception is paused.

Since ConSSTEP possesses two sets of configuration registers per component, the processor may forgo the need to save state by using this "pausing" technique on the exception cycle, while configuring the alternate set to handle the exception. Since the engine is not configured for the exception handler, it incurs a standard latency penalty of 10c.c. The scheduler must also save the alternate set's bundle ID which is overwritten by the handler so that the bundle may be restored once the handler has finished executing. Such a technique allows ConSSTEP to implement precise exception handling with less complexity in comparison to conventional processors, where ConSSTEP suffers a configuration cost at the advantage of avoiding pipeline flushing and state restoration. Likewise other threads may concurrently continue to execute in their respective engines, where single-thread performance remains unaffected.

Since temporary data may be present in the rS latches when an exception is raised, handler bundles are generated without the ability to temporarily store values in rS units. Therefore handler bundles are formed under the assumption that the rS units are only used for data propagation, where the EARF is responsible for any operand writes and/or reads. Under such an assumption, it is also possible that the handler may require value(s) which are temporarily latched and not written back to the EARF during the time an exception is raised. To accommodate such cases, the datawidth of the interconnect wires are widened by 5 bits, where the most significant bit signifies data validity, and the next 4 bits represent the value's writeback register should an exception occur. Thus during a handler's configuration process, the engine simultaneously performs a writeback phase which allows each individual row to write out (valid) temporary values to the EARF per cycle (i.e. first cycle bottom rS latches, 2nd cycle the last row of ALUs, the next cycle the 2nd last row of rS latches etc), where input values required for the handler are read after the configuration process.

Once the handler has completed execution, the engine sends a completion signal to the scheduler. When the system is verified as stable, the scheduler sends a signal back to the paused engine to continue execution, where the configuration register set which handled the exception is also reconfigured to its previously predicted bundle re-sent by the scheduler. This technique of engine "pausing" is also used to maintain synchronization between thread barriers, monitored by the scheduler.

Note that the concept of exception handling presented here may also be applied to interrupt handling if required.

5.9 Scheduler

The scheduler plays an essential role in the ConSSTEP core. Specifically, it is responsible for dynamically monitoring engine execution (i.e. n-threads simultaneously executing) and acting according to each engine's state. Branch prediction in ConSSTEP assumes identical functionality to conventional processors. Therefore although engines execute bundles created statically by the PhysC, the branch outcome must be resolved dynamically by the scheduler to verify whether the taken/not-taken branch outcomes per thread were predicted correctly. Hence the scheduler (per thread) must 1) implement branch prediction, 2) determine the next bundle ID to send to the engine based on the taken/not-taken prediction for engine pre-configuration, and 3) verify branch prediction outcomes. It must also dynamically monitor and handle any exceptions raised across the thread workloads.

In the cold case of scheduling, an initial bundle ID is generated by the PhysC and provided to the scheduler (per thread) for commencing execution. Since all bundles are terminated on conditional branches (or indirect unconditional) using an aggressive pipeline, branch predictions must occur on the fly. Therefore the scheduler possesses a branch predictor per thread which predicts the outcome of the executing bundle's branch instruction, where the predicted bundle ID is sent to the engine for pre-configuration. In order to distinguish bundle ID's based on taken or not-taken branch outcomes, each thread in the scheduler possesses a 2-way direct mapped cache so that the current bundle ID may be indexed and its respective taken/not-taken (predicted) bundle ID may be obtained.

The scheduler must also monitor flags raised. In particular during regular execution, the scheduler must monitor branch flags sent back by the engine indicating bundle completion and the branch outcome (0 = not taken, 1 = taken). Thus if predicted correctly, the scheduler's branch predictor uses its history to predict the next bundle's outcome, indexing and sending out the next predicted bundle ID for the engine's pre-configuration as the correctly predicted bundle continues to executes. However if a misprediction occurs, the scheduler must 1) send out a stop signal to the engine to halt its execution, 2) verify that the misprediction is not

a loop, 3) correct its branch history given the misprediction, and 4) send out the correct bundle ID if required.

If the prediction was in fact a loop, this signifies that the correct bundle currently resides in the configuration set which just finished executing. Therefore the scheduler sends out a "loop" signal to the engine so that new external values may be read into the RRB and the multiplexer bit in the engine's structures are redirected to execute from the previous configuration register set. Once these steps have completed and the branch predictor's history is corrected, the newly predicted bundle may be sent for pre-configuration as the loop executes. In the case of a total misprediction (i.e. both configuration register sets are incorrect), the scheduler must send a "clear" signal to the offending engine so that its contents may be cleared, where the correct bundle ID is then sent to be configured (undergoing a 10 clock cycle configuration penalty in total). Once the branch history has been corrected, a prediction is made on the next bundle to be executed, which is send out for pre-configuration as the corrected bundle executes.

Finally the scheduler must also monitor any exceptions raised considering a general purpose processing environment. In the case of an exception, the scheduler sends out the "exception" flag, thereafter indexing the respective exception vector and ID to be sent to the engine for handling the event. The scheduler then waits for the engines completion signal to verify the exception was handled and a normal processing state has resumed. The scheduler may then send a signal back to the engine to continue from the paused cycle in the alternate configuration register set.

Although the scheduler is not very complex in design, it is of a unified nature and therefore must monitor n-threads simultaneously, updating/verifying each engines respective predictions, exceptions etc. Its hardware must also integrate direct map caches for indexing taken/not-taken bundle predictions per thread. Thus it is expected that a unified scheduler approach will require an increase in logical complexity as threads scale in ConSSTEP, especially in terms of IO ports and storage requirements. Hence the design and implementation of a distributed scheduling unit remains future work for ConSSTEP scalability.

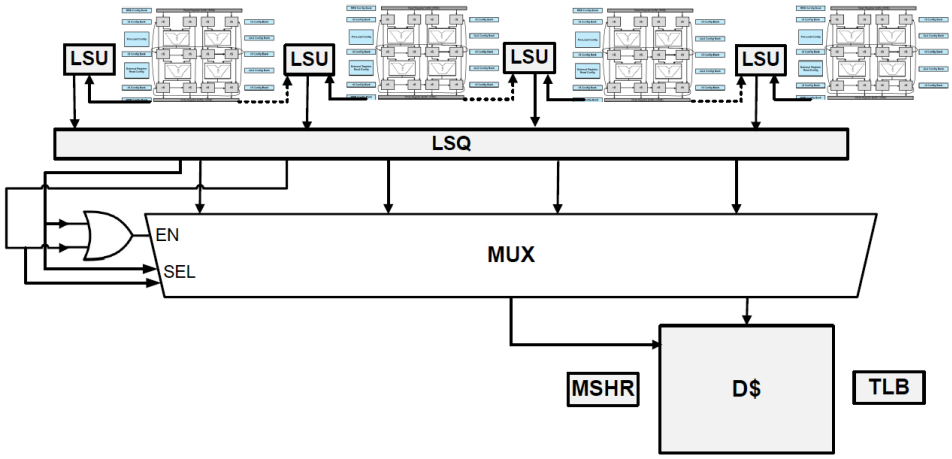


Figure 5.7: ConSSTEP Cache System (to match conventional CPU model)

5.10 Data Memory Accesses

As ConSSTEP may support up to n engines operating concurrently, the data cache may also be required (in the worst case) to access up to n words, possibly belonging to n unique cache lines and/or virtual pages simultaneously. Consequently, the cost of adding n ports to the data cache may not be feasible in terms of energy and/or latency per memory instruction depending on the number of engines per core. Therefore, ConSSTEP employs a banked data cache, where banking is performed at the cache line granularity similar to conventional SMT models. Although engines are configured to access different banks, they may also access the same element for certain (and/or atomic) data operations. In the case of a bank conflict, the memory controller randomly selects which thread may access the cache, while the other thread/engine data is buffered in the crossbar.

ConSSTEP employs a conjoined system between LSUs and the engines so that the varying memory demands of the threads may be managed fairly, whoen seen in Fig. 5.7. For instance, if an engine requires two memory requests while the other engines make no requests, such a memory system may sufficiently tolerate these varying demands. Specifically, in a 2-Thread (2T) ConSSTEP configuration, two LSUs are assigned to each engine, where the 4-Threaded (4T) configurations are assigned one LSU per engine and are shared between engines as seen in Fig. 5.7. Signals sent to the shared LSUs are managed by WRBs and RRBs, where the LSQ buffers pending memory requests and manages cache accesses. If two engines contend for a shared LSU, one engine will be stalled until the load/store instruction is

resolved in the alternate engine (using a round-robin algorithm).

The memory system for both the conventional processor baselines and ConSSTEP were designed with the same configurations (ports, banks etc) to simulate identical data memory system. This framework allows the architectures to be directly compared without memory related interference, and for ConSSTEP to achieve identical memory accesses latencies. Therefore, any additional LSUs present in the ConSSTEP architecture in comparison to the baseline are multiplexed (similar to Fig. 5.7) to maintain the same memory parameters as the baseline processors. Since the same memory system is used by ConSSTEP, the respective bottleneck properties typical of conventional memory system hierarchies are not eliminated, and therefore also endured by the core. Thus the purpose of ConSSTEP is purely to improve upon the performance of the processor itself considering the same memory limitations of conventional CPUs.

Chapter 6

Compilation

6.1 Introduction

This chapter outlines the specifications of ConSSTEP's compilation process, from benchmark input to configuration data output. Accordingly, this chapter discusses the concept of a logical compiler (a term coined in this thesis for the concept of a conventional compiler) and respectively efficient target ISAs for such a compilation approach. The chapter then outlines the Physical Compilation (PhysC) process introduced in this thesis, and specifications pertaining to thread bundle formation, inter- and intra-bundle dependencies, and its respective mapping algorithms which are tested for performance and energy efficiency in Chapter 8. The chapter then discusses the PhysC's use of sequence graphs which are used to map instructions and bundles temporally and spatially throughout the ConSSTEP core. Thereafter the generation of configuration data is discussed which is used to execute the thread workloads of a multi-threaded application.

6.2 Logical Compiler

A logical compiler is a standard compiler used by modern computing systems. Any compiler may be chosen for ConSSTEP, however physical compilers benefit from RISC-like target architectures. Specifically, three-operand architectures (i.e. typical RISC ISAs such as ARM) allow destination registers to be distinct from their source registers, providing more renaming and execution flexibility, and most importantly fixed length instruction decoding. Conversely, two-operand architectures (i.e. typical CISC-like ISA's such as x86) use one of their operands as both a source and

destination register, and possess variable instruction length decoding [70]. Therefore CISC-like formats impose greater difficulties for extracting register and instruction dependencies, leading to more complex designs for the next compilation phase. Regardless of such issues however, the compilation process may be implemented for any target ISA.

For this work, the ARMv7 ISA was selected for both the physical and logical compiler target architecture (other ISAs remaining future work) with a GNU-based gcc compiler. Therefore the logical compiler may compile any C, C++, and Fortran language while supporting a variety of programming models as previously discussed in Chapter 3.

6.3 Physical Compiler (PhysC)

A PhysC is the joint effort between compiler designers and computer architects, used for promoting a smarter compilation process and maintaining compiler, software, and binary compatibility. A PhysC's main objective is to obtain a compiled binary and perform macro-processing, generating bundles of microcode logic to execute a given application on an underlying configurable architecture. For this work, input traces are provided to the PhysC as the focus of this work is primarily on the feasibility of the ConSSTEP architecture and the general compilation flow required to support such an underlying architecture. Consequently in the case of memory references, disambiguation is handled easily (in addition to other phases such as alias analysis), which allows for a less complex PhysC design. Similar traceless translation processes have also been proposed [44, 48], demonstrating that such a compatible binary translation process is feasible. For this thesis work, the PhysC invokes trace-based input to mimic steady-state behaviour. The overall physical compilation flow is shown in Fig. 6.1.

6.3.1 Bundle Formation

Bundle formation is the first stage of physical compilation. A bundle is defined as an entity which is identified during physical compilation. Specifically, a bundle is a set of inter-related instructions extracted from an instruction stream, terminated once a conditional (or indirect unconditional) branch is encountered. For this work, the

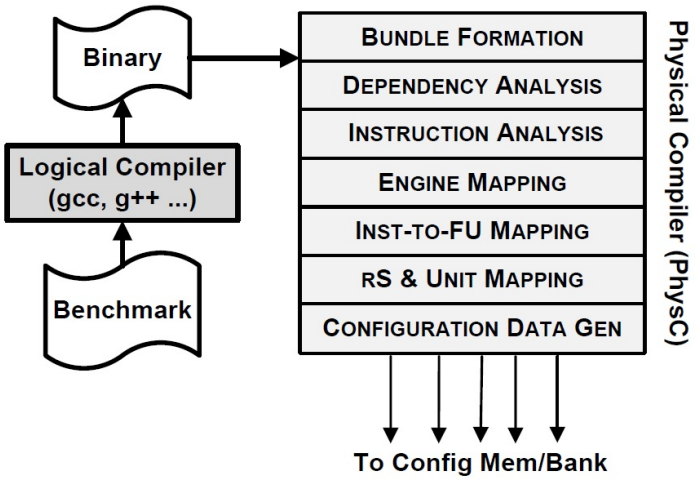


Figure 6.1: ConSSTEP's PhysC Flow

Gem5 simulator [71] was used to obtain instruction traces for various multithreaded workloads, where the traces were provided as input to the PhysC. The PhysC then creates bundles based on the provided instruction streams per thread, following thread traces until bundle terminating branches. The instructions collected prior to a given branch are labelled part of the same bundle, where unconditional (direct) branches may be eliminated since they are simply required for redirection. Once a conditional branch is reached, the taken and not taken bundle IDs are generated and saved for inter-dependency analysis. In this work, exception handlers and their respective vector addresses must be directly provided to the PhysC for handler bundle generation.

Since the ARMv7 ISA is employed in this work, conditional instruction suffixes (i.e. branch elimination for smaller If-Then statements) may be generated by the compiler. This is dependent on the level of compiler optimization, and referred to by ARM as IT blocks (i.e. `ITE` (`If/Then-Else`) for one `if` and `else` statement in the code, `ITTE` for two `then`, and one `else` statement etc). The suffix appended to each assembly instruction dictates whether the instruction belongs to the IF or ELSE block. Such suffixes are analyzed by the PhysC and inserted into the appropriate taken or not-taken bundle. This process occurs for all application bundles.

Once a conditional branch is reached, Gem5's predicted branch route is followed by the simulator, where the PhysC continues to create new bundles with the provided trace. Gem5's predicted route (i.e. taken or not taken) is stored as metadata to

the previous bundle and assumed correct during trace replay in the PhysC. When a bundle/branch misprediction is encountered however, Gem5's simulation stream is redirected in Gem5. Hence in this case of a misprediction, the PhysC must:

- Stop the current bundle's formation
- Adjust the previous bundle's prediction to the correct one
- Start bundle formation with the correct route's bundle.

Bundles which are mispredicted yet executed by ConSSTEP (but not traced by Gem5) are filled with few dummy instructions that the engine may execute and clear once the correct branch outcome is determined.

6.3.2 Data Dependency Analysis

Bundle formation allows the PhysC to generate a dependency graph, while also illustrating the dependencies between both taken and not taken bundles. This step allows the PhysC to extract inter-bundle dependencies, which are used to temporally sort the application bundles at a coarser-granularity to implement branch prediction routes. The PhysC also checks for bundle duplication at this stage so that configuration time may be reduced and optimized, with less memory storage required in the backend.

6.3.3 Instruction Analysis

After generating bundles and determining inter-dependencies, the PhysC assesses intra-dependencies between bundle instructions, eliminating false dependencies (i.e. Write-After-Write (WAW) and Write-After-Read (WAR)), and extracting producer-consumer dependencies without the need for explicit hardware register renaming. Since no *physical* register file exists in ConSSTEP to limit the number of register renames permitted, the PhysC is free to rename operands as necessary with no hardware restrictions, while keeping track of final *architectural* register values to be written to a thread's EARF. The compiler must also keep track of instruction types (i.e. integer, FP etc.) and its associated dataflow between types. This phase in general allows ConSSTEP to extract and transform a restricted dataflow binary (i.e. logically compiled) to a dataflow-like executable on its engines. Memory dependency orders are also analyzed and maintained temporally as imposed by the ISA's memory model.

6.3.4 Engine Mapping

To appropriately assign thread bundles to an engine, the bundle's respective instructions and inter-bundle dependencies are analyzed and compared to the available (and ideal) execution engines present in the core. Contrary to Single-Instruction Multiple Thread (SIMT) models (i.e. GPUs) which assign threads to identical SIMD units for execution, ConSSTEP supports various engine sizes. Thus inter-bundle placement may be quite impactful on increasing ConSSTEP's performance. For instance, placing a computationally-intensive bundle on a small engine may affect performance negatively in comparison to mapping the same bundle to a larger engine which can support wider issue widths and more temporary data storage.

Accordingly, a set of mapping algorithms will be assessed during Chapter 8 to determine engine-to-bundle assignments and their effect on performance and energy efficiency. The following three bundle placement algorithms are used in this study:

- **First Come First Serve (FCFS):** Assigns the first ready bundle to the first available engine.
- **Random (Rand):** Selects a random (available) engine for a given bundle's execution.
- **Demand:** Considers the total number of instructions per bundle, the computational intensity of the bundle, and the max number of instructions that may execute at once. The bundles are then classified and assigned to the most suitable engine, if available. Else, the method will search for the next suitable engine for the bundle's execution.

To maintain low hardware complexity, once all bundles have been mapped to their preferred engine, bundles of a thread are assigned to one specific engine according to the *average* preffered engine mapping per thread.

6.3.5 Instruction-to-FU Mapping

Once all threads and their bundles have been assigned to an appropriate engine, the next PhysC stage maps instructions of a bundle to FUs according to the producer-consumer dependencies generated and/or any FU type dependencies based on the instruction's operation. All instructions are time sorted according to dependencies and the availability of FUs. Thus if FU conflicts exist, the instruction is scheduled

for the next clock cycle. This phase of the PhysC is also responsible for exploiting ILP within the bundles, using a maximum issue width dictated by the number of FUs present in the given engine. If the instructions scheduled for that cycle exceed the engine's issue width, they are then scheduled for the next clock cycle with all its dependencies scheduled thereafter. In such ILP exceeding cases, priority is given to instructions with a higher number of dependencies so that such instructions and their future dependent instructions may execute as soon as possible.

As results produced in an engine may also traverse any number of rS hops to its consumers in a single cycle, dependent instructions experience combinational logic delays. Thus majority of the effects regarding intra-bundle placement arise in energy efficiency by reducing hop counts between dependent instructions. Accordingly, this work employs the following four instruction placement algorithms which are analyzed for efficiency during Chapter 8

- **Static-Filler (Static):** Assigns instructions to the engine's FUs in a round-robin fashion until completely filled, and then starts assigning instructions to the first FU once again in the next time slot if the issue width is exceeded. Instruction belonging to the next clock cycle are then filled in the same manner, starting the first FU etc.

- **Dynamic-Filler (Dyn):** Maps a bundle's instructions to FUs considering data dependencies. The method places dependent instruction in the first available FU found in the row directly below the producer instruction, or nearest row if the subsequent row is not available. If no dependencies exist, the instruction is placed in the first available FU. Therefore a direct consideration for reducing hop count between dependent instructions in the main grid is the objective of Dyn.

- **Depend-Filler (Dep):** Maps instructions in a similar manner as Dyn, however giving top priority to external unit dependencies. That is, if an instruction has an external unit dependency, it will be placed in the bottom rows of the engine to minimize hop count to the WRB and external unit. If no FUs are available, the next best case FU will be considered etc. Dep then gives second priority to the method invoked by Dyn.

- **Random (Rand):** Assigns each instruction randomly to a FU in the assigned engine. In order to reduce mapping time, if a free FU has not been selected

after 3 tries, a reverse Static-Filler approach is applied i.e. search the engine's last FU first, and decrement thereafter to find first available FU.

It is important to note that load, store, and/or any external unit instructions within a bundle are extracted prior to this step as these instructions must execute on external units.

6.3.6 rS & Unit Mapping

Once instructions have been successfully mapped to their respective FUs, the PhysC then evaluates the mappings to determine how each rS unit and engine structure should be temporally configured, considering the data transport of producer-to-consumer dependencies and the temporal/spatial placement according to the FU mapping determined previously.

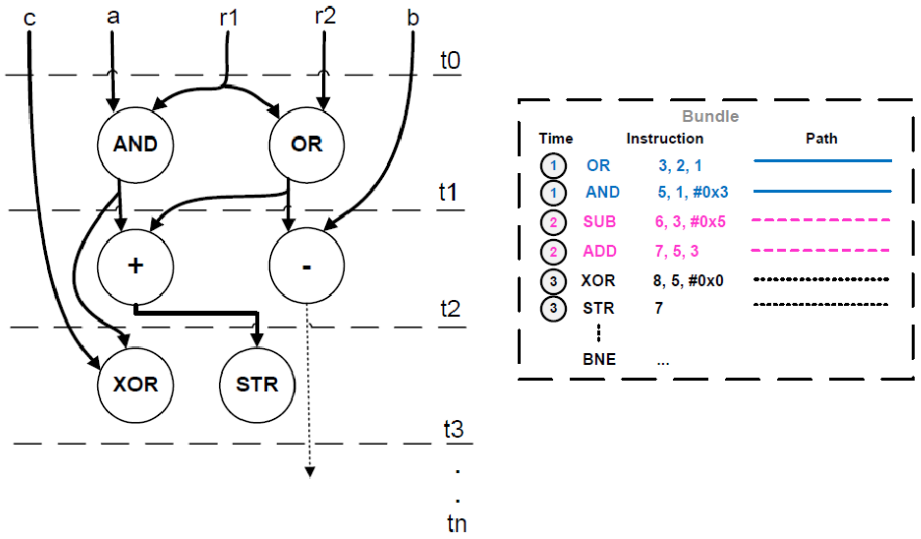


Figure 6.2: Sequence Graph of an Instruction Bundle

An example of the process used by the PhysC to determine control logic per component is demonstrated in Fig. 6.2 and 6.3, adapted and altered from the principles of Reconfigurable Computing Systems presented by L. Kirischian [5]. Fig. 6.2 displays a sequence graph (SG) of the bundle provided in the execution example of Chapter 3.6. Such a SG is representative of bundle's data flow and control sequences during computation, using the information derived from previous PhysC stages. Specifically, the vertices of the SG represent the functionality required by the engine to process an instruction at a given time, whereas the arcs display the

data dependencies between the instructions. The SG therefore represents a polar and acyclic graph for a given bundle, vertically ordered with respect to increasing time [5]. External inputs are illustrated at t0, whereas final outputs are represented by a *dashed arrow* as displayed in the example above.

Since the preceding "Instruction-to-FU" PhysC stage determined the FU mappings per instruction (referred to as resource binding in RCS), this stage is responsible for determining the interconnect propagation/latching, and engine structure mapping based on such information. The basic mapping scheme used by the PhysC is shown in Fig. 6.3, where the constants a, b and c are high level representations of the immediate values found in the bundle's instruction stream (0x3, 0x5, and 0x0, respectively), with the FU mappings are displayed accordingly.

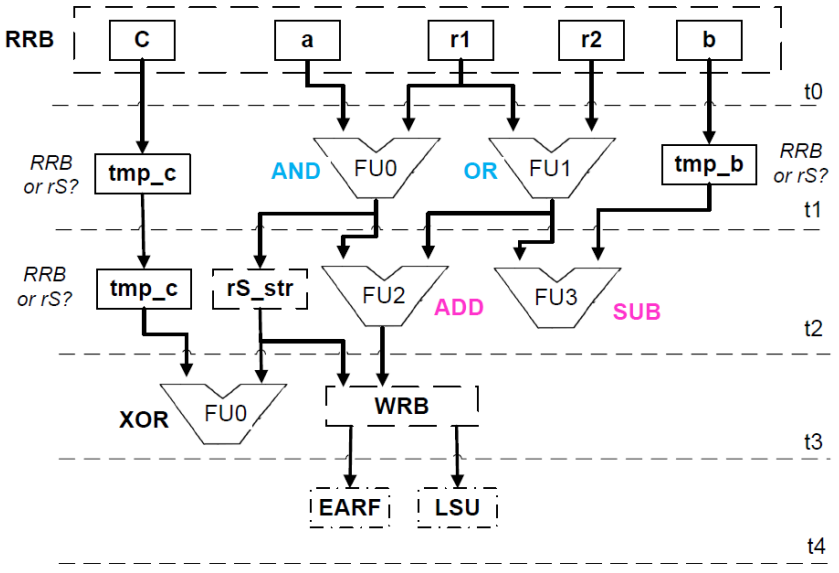


Figure 6.3: Scheduled Sequence Graph with Resource Binding, Data Latching and Transport

External inputs of engines are latched by the RRB. It is possible however that the RRB must output other external data from the same port at a given. To accommodate this possibility, values such as c and b must be temporarily latched in the SG representation so that subsequent RRB values may propagate to their consumers. Such values are labelled "tmp_" and may be either latched by an rS or buffered by the RRB if no other data is required for output in the next cycle. Such latching strategies are determined by the PhysC during this stage. On the contrary, the r2,

r1, and 'a' values do not require temporary latching as they are consumed immediately. In the case of the AND instruction, the result requires both propagation and storage as previously discussed, and therefore such cases are marked as "rS_str" in the SG so that the PhysC may determine the best rS unit for mapping. Finally, external outputs represented on the SG are latched by the WRB (represented as a unified latch in the figure), where the external units are also included. The external FUs are represented on the SG as well so that the PhysC may generate WRB request signals to interface the units on the given cycle.

Once all components and latching requirements have been determined, the PhysC may then route data according to the timing schedule derived from the binded SG. For instance in the case of the RRB and its constants c and b, since no values are required for output within the next two clock cycles, the immediate value 'c' may be buffered by the RRB until clock cycle 3, where it may thereafter be propagated directly to FU0, with a similar case for 'b'. Detailed timing requirements for each component are then derived, with an example of the FU, RRB, and WRB timing requirements of Fig. 6.3 displayed in Fig. 6.4. As seen in Fig. 6.4, FUs are assigned their operation per clock cycle during this stage, and similarly the RRB and WRB are also assigned their input/output values per port per clock cycle. Thus the PhysC takes this information and generates control logic for each engine structure's and configuration latch. The EARF input registers, immediate values, and final EARF output registers are also determined during this time.

FU 0	
Time	Operation
0	NOP
1	AND
2	NOP
3	XOR
....	...

FU 1	
Time	Operation
0	NOP
1	OR
2	NOP
3	NOP
....	...

FU 2	
Time	Operation
0	NOP
1	NOP
2	ADD
3	NOP
....	...

FU 3	
Time	Operation
0	NOP
1	NOP
2	SUB
3	NOP
....	...

RRB				
Time	Output			
0	r2	r1	r1	a
1	c	-	-	b
2	c	-	-	b
...	...	...	...	...

WRB								
Time	Input				Output			
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-
3	-	tmp_rS	FU2	-	-	-	-	-
4	-	-	-	-	-	EARF	LSU	-
...	...	..	..	...	...	...	...	...

Figure 6.4: Timing Schedule for Engine Structures

In the case of the rS units, such programmable logic is slightly more difficult to generate. Specifically, the rS propagation and/or temporary storage details specified in the SG must be determined. In the case of a temporary storage, the value

may be stored to any route-able and available latch. Propagation however must strictly follow an xy routing protocol to manage PhysC complexity. In the case of storage conflicts and/or contention during data transport, the PhysC may resort to re-routing in the case of propagation, alternative storage locations in the case of temporal latching, or in the worst case delaying execution by a clock cycle if neither option is possible. All such rS propagation and storage mappings are then determined. More detailed studies regarding contention and utilization during the rS mapping phase is elaborated during Chapter 8. With regards to FP engines, since such engine types do not possess rS units, the FP RRB, WRB, external register reads/writes and data flow configurations between unit types must also be considered in a similar manner.

6.3.7 Configuration Data Generation

Once all structures and routes have been successfully mapped and determined, the configuration data needed by ConSSTEP engines are generated for each rS, ALU (opcodes), RRB, WRB, and external memory access based on the information provided in the timing schedule of the previous stage. All configuration data is then passed to its respective banked memory and the scheduler.

6.4 Summary

This chapter discussed the details of the logical and physical compilation process used in this work, from application input to configuration logic generation for the underlying architecture. Now that the ConSSTEP architecture is well understood, the experimental methodology may now be presented, which is used for experimental testing, results, and analysis.

Chapter 7

Experimental Methodology

7.1 Introduction

This chapter outlines details pertaining to the experimental methodology used to obtain the results presented in the next two chapters. Specifically, this chapter outlines various details of the architectural framework/performance modelling and physical modelling (latency, area and power/energy) used to elaborate the ConSSTEP architecture and all baseline processors.

This chapter opens with the architectural framework section, which discusses details of the simulator and all processor core specifications used during testing and evaluation. The benchmarks used to evaluate the cores are then discussed, where physical modelling specifications are provided thereafter. Specifically, physical modelling discussed details of hardware logic area, its respective wire delay, and cycle time.

7.2 Architectural Framework

7.2.1 Simulators

To validate ConSSTEP's performance, an in-house simulator with trace-driven execution was used to simulate the core shown in Fig. 7.1. The simulator accepts benchmark thread traces as input (logically compiled), and a data file specifying the the number of threads per core and the configuration of the ConSSTEP core, i.e. x-FUs by y-FU engine, x_1-FU by y_1-FU engine etc. The traces are then processed by the PhysC, where the simulator assembles the core configuration specified

by the data file for both ConSSTEP and the baselines. The bundles are then assigned to their respective distributed Configuration Caches (C-Caches), where the scheduler commences simulation. Once simulated, the simulator outputs various performance statistics, used in Chapter 9.

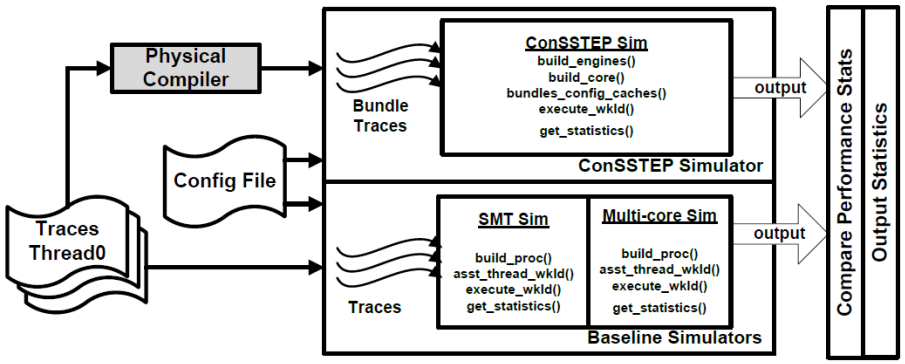


Figure 7.1: In-house Simulator Framework

Although an in-house simulator was used to assess performance, the Gem5 [71] simulator was also used to generate the traces using Full-System (FS) mode to obtain various execution details required by the ConSSTEP simulator. Since Gem5's SMT simulation was problematic for both System-call Emulation (SE) and FS mode at the time of this work, a trace-driven simulator for both the SMT processor and the Single-Thread Core (STC) multi-core were also developed. All traces were obtained using the Linux aarch-ael image, and m5op definitions to instantiate application checkpoints, loaded onto the virtual file system consisting of n-processors, where each processor runs one thread's workload for the n total threads. Thereafter the traces were collected and provided to the PhysC. The ARM ISA was selected for both the physical and logical compiler target architecture, using GNU-based gcc.

The trace-based, in-house simulators for both ConSSTEP and the baselines were coded in C++, running on a Ubuntu OS and the Intel Core i7-5820K CPU at 3.30GHz with 8GB of RAM. Details of each simulator are provided in Table 7.1. The simulators specifically were designed for the architectural comparisons of a single ConSSTEP core, a SMT core, and a multi-core system consisting of monolithic STCs (i.e. one thread/core). Since the ARM ISA is used, engine configurations for the baselines were modelled on a plausible SMT version of the ARM Cortex-A57 architecture, replicating the branch predictor, Instruction Fetch, rename logic, and

physical register file per thread, where other structures are shared as outlined in Table 7.1. Accordingly, this work considers 2-thread(2T) and 4-thread(4T) multi-threaded models during experimental testing, which were imposed by maximum die size constraint of the SMT model.

The baseline simulators (the Single-Thread (multi-)core and SMT simulator) consist of an 8K Bi-Modal branch predictor with 2K Branch Target Buffer (BTB) per thread. The specifications of the baseline CPUs assume a Harvard architecture, 12-stage front-end, a 2 clock cycle issue-to-dispatch latency, 3-wide instruction fetch and decode, 8-way issue consisting of 64-entry IQ, 36-entry Physical Register File (PRF), 128-entry FP PRF, 4-wide commit and ROB size of 128 entries. The backend FUs consist of 2 ALUs, 2 LSUs, 1 complex Integer unit, 2 FP units, and 1 branch unit. Evaluations for all baselines consider a multi-core environment by simulating the processors using a throughput-limited DRAM memory of 2GB/s per core, representative of current multi-core systems [35].

Several configurations for the ConSSTEP architecture were also tested, as detailed in the forthcoming chapter. Of the FUs specified in the data input file, half the FUs are assigned complex integer functionality by the simulator, where the rest are assigned basic ALU functionality. As previously mentioned, LSUs are conjoined between engines for 4T configurations, where 2 LSUs are assigned per engine for a 2T configuration. Each engine also possesses a conditional branch unit and branch predictor per thread, with a respective 2-FU FP engine. The memory system for ConSSTEP also assumes the same throughput-limited DRAM memory of 2GB/s per core, where *all cores*, including the baselines, invoke a 32KB 4-way L1 D-Cache with 64B lines, and 64-entry dual-port TLB. A sample 2T $\langle$4x6$\rangle$ ConSSTEP architectural layout is provided in Fig. 7.2.

Locks and other thread synchronization primitives require special semantics within the simulator to handle a multi-threaded trace-driven approach. Specifically, all calls to synchronization primitives in the benchmarks were recorded and enforced within the simulator's scheduler during replay. Therefore both the ConSSTEP and baseline core simulators invoke execution-driven simulation for synchronization primitives, while maintaining a trace-driven approach for the standard execution of traces.

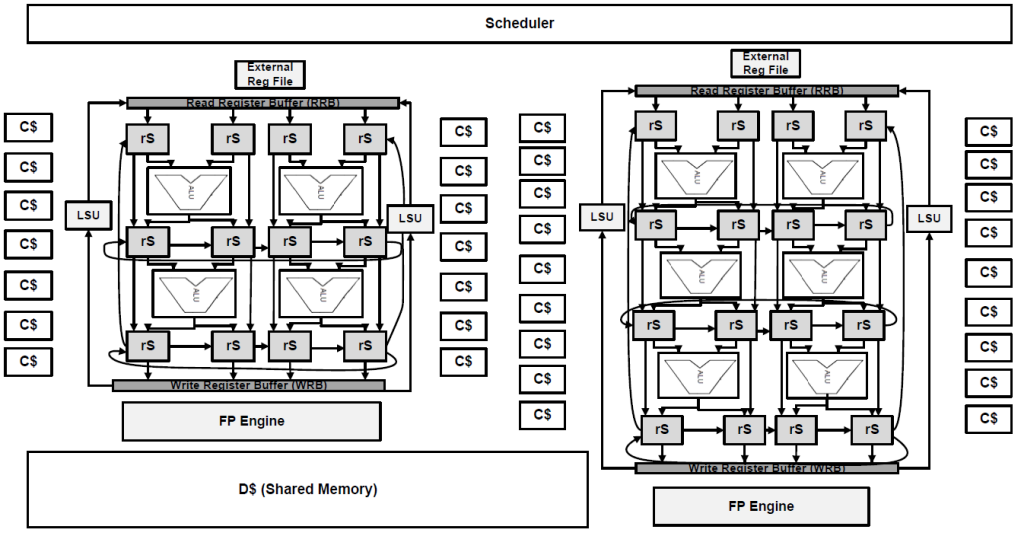


Figure 7.2: ConSSTEP 2T ⟨4x6⟩ High-level Core Layout

Table 7.1: Simulation & Modelling Parameters

Feature	**ConSSTEP**	**Baseline, OoO SMT/STSC ARM CPU**
Branch Predictor	8K Bi-Modal Branch predictor, w/ 2K BTB entries, per thread	
Functional Units	Per n-FU engine: n-ALUs (1 incl complex INT), 2 LSU/engine (2T)/shared for 4T, 1 cond branch, 2 FPU/FP engine LD/STR conflicts buffered in cache crossbar	2 ALUs, 2 LD/STR, 1 complex INT (4c.c) 2 FPU, LD/STR conflicts buffered in cache crossbar Modelled according to ARM Cortex-A57 architecture with SMT multi-threading
Execution Parameters	nxm INT engines (T-Threads). 1x2 FP engines At least one mul/div per engine 16-entry int & FP ARF per engine Branch Predictor per engine	3-wide fetch, 4-wide commit, 64-entry IQ (i.e. 8-way issue) 12-stage front-end, 2c.c issue/disp, squash recovery 36-entry PRF, 24-entry FP PRF, 128-entry ROB **SMT Replicates:** Branch, IF/PC, Rename, PRF
Memory System	32KB D$ 4-way, 64B line (2c.c) 64-entry dual-port TLB Total aggregated 746KB C$ (max bank = 64KB) L2 D$, 4MB 16-way 12c.c, miss lat. 215 c.c	32KB D$ 4-way, 64B line (2 c.c) 64-entry dual-port TLB 32KB I$, 2-way, 64B line (2 c.c) L2 D$, 4MB 16-way 12c.c, miss lat. 215 c.c

7.2.2 Benchmarks

Table 7.2: Benchmark Descriptions

Benchmark	**Common Characteristics (Type, L1...)**
Barnes	INT, SPr, low L1 miss rates
Blackscholes	INT, SPr, low L1 miss rates, high bank conflicts on reads, high branch count
Fluidanimate	INT, SMT, low L1 miss rates, high branch count
Cholesky	INT, SMT, fair shared L1/L2 accesses
OceanNP	FP+INT, SMT, low L1 miss rates, computationally-intensive
Swaptions	INT, SMT, low L1 miss rates and shared accesses
FFT	INT, SMT, high shared L1 reads and bank conflicts
Streamcluster	INT, SMT, high L2 miss rates
Water-Spatial	FP+INT, SMT, low L1 miss rates, computationally-intensive
LU	FP+INT, SMT, high L1 shared write accesses, computationally-intensive

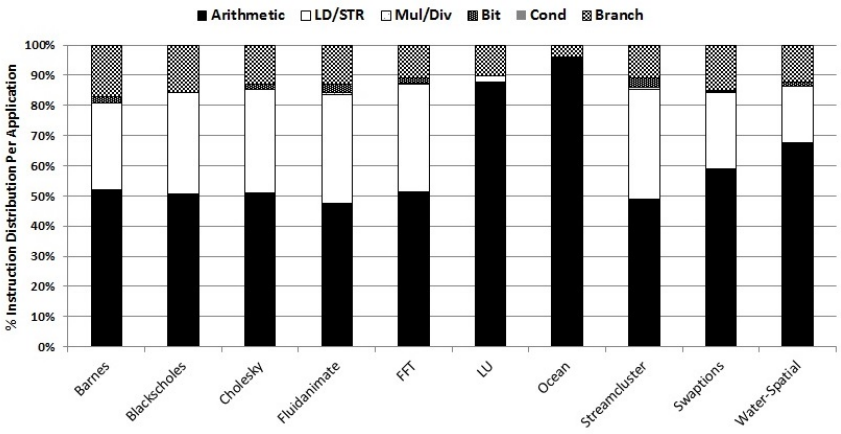


Figure 7.3: Instruction Distribution for all Benchmarks

As ConSSTEP's main objective is to improve the performance of multi-threaded workloads, we evaluate all processor models using the multi-threaded PARSEC [12] and SPLASH-2 [13] benchmark suites. Specifically ten benchmarks with different characteristics are used to evaluate ConSSTEP: Barnes, Blackscholes, Fluidanimate, Cholesky, Ocean NP, Swaptions, FFT, StreamCluster, Water-Spatial, and LU. Fig. 7.3 outlines the instruction type distribution for all the benchmarks, where Table 7.2 specifies the predominant memory and multi-threading characteristics for each benchmark. Benchmarks which specify SMT workloads generally consist of multiple

threads where each thread is assigned a specific workload. Conversely benchmarks labelled as Speculative Precomputation (SPr) are multi-threaded workloads consisting of one or more helper threads which solely provide data-prefetch, while the other thread(s) compute(s). Such SPr benchmarks therefore signify more memory-intensive benchmarks.

7.3 Physical Modelling

To assess factors of latency, area, and power consumption, hardware structures for all processors were modelled using a combination of synthesis and CACTI 6.5 [72]. Structures that required small storage (reservation stations etc), or structures which were more customized in nature (i.e. rS units) were modelled using VHDL and synthesized using Synopsys Design Compiler at the 45nm technology node with the OpenPDK library (NCSU/OSU). Larger structures consisting mostly of SRAM (such as the caches etc) were modelled using CACTI also at the 45nm node. The following sections outline the various physical modelling details used to assess and validate the ConSSTEP architecture during experimental results and analysis (Chapter 9).

7.3.1 Area Estimates

During experimental testing, this work considers the rS interconnect wiring dimensions for all engine types (integer and FP), laid out horizontally and vertically. Metal layer 4 was selected for the rS column wiring, and layer 3 for the vertical wiring. Following the 45nm library layout rules, metal layer 3 and 4 both required a minimum spacing width of 0.14um. The total width for the wiring of the integer engines considering 4-, 6-, and 8-FUs (and the *longest torus wire*) were 134.8um, 218.9um, and 269.6um respectively. Similarly, the total width for the wiring of a 2-FU FP-based engine was approximately 265.0um. The total wiring widths for both types of engines were therefore less than the square root of their respective engine areas (see Table 7.3). Based on these results, ConSSTEP's area in all cases is not wire-dominated.

7.3.2 Wire Delays

Wire delays for architectural modelling were calculated using the distributed RC model, $1/2\mathrm{R}_{wire}\mathrm{C}_{wire}$ [73]. For metal layer 3 with the worst case dimensions of

134.8um x 0.14um (i.e. longest torus wire), the delay was approximately 0.174ps for a 4-FU engine, 0.458ps for the 6-FU engine, and 0.872ps for the 8-FU engine. These results were calculated according to the parameters specified by the 45nm OpenPDK library, considering R_{wire} equal to 0.25 ohms per square distance, and C_{wire} 0.00015pF per square distance. According to these results, such single-cycle wire delays are therefore considered negligible during cycle time calculation. In general, wire delay issues were mitigated by using short wires between rS units while also considering the longest torus connection which is also negligible as seen in the calculations provided here (and further enforced by restraining engine size).

Although the rS interconnect adopts Network-on-Chip (NoC) methods, data may traverse a maximum rS unit critical path length per engine and such wire delays must be considered. Although the wires are kept short using the rS interconnect as shown above, wire repeaters [74] are inserted at every rS unit to maintain the signal speed and integrity of NoCs. Thus considering that data may traverse a critical path length, and that there are three gates per multiplexer, two multiplexers per rS, and one gate per repeater, a 2-FU engine will experience 36 gate delays (gd), a 3-FU engine 39gd, a 4-FU engine 42gd, a 6-FU engine 48gd, and an 8-FU engine 54 gd. Assuming 4ps per gd at the 45nm node[74], there is an approximate 24ps difference in critical path delay per engine configuration, demonstrating ConSSTEP's scalability in terms of frequency. Thus the two extremes of a 2-FU engine and 8-FU engine have a maximum wire delay of 0.14ns and 0.216ns, respectively, and considered negligible.

Table 7.3: Delay, Energy per Access, and Area Results

	ConSSTEP			**Baseline SMT OoO CPU**		
Component	**Delay (ns)**	**Energy (pJ)**	**Area (um^2)**	**Delay (ns)**	**Energy (pJ)**	**Area (um^2)**
2T/4T Scheduler vs Fetch	0.37/0.94	1.01 / 4.0	584480 / 2450000	1.84	4.66	27831.4
Decode Unit (3-way)	-	-	-	3.01	4.19	9468.6
Rename (3-way)	-	-	-	3.28	10.41	31125.8
ROB	-	-	-	0.53	1.101	204300
8-entry IQ	-	-	-	2.32	4.348	18961.6
Config Reg Write(64b)/Read (1b)	2.18/0.01	1.16/0.01	3917.2	-	-	-
Payload RAM	-	-	-	0.70	2.611	19926.0
INT ARF vs PRF	1.27	2.54	18680	1.37	16.52	135254.4
FP ARF vs FP PRF	1.89	8.4	41370.2	2.1	23.77	146278
ConSSTEP INT ALU vs ALU+Bypass	1.48	1.13	7869.69	3.50	10.15	48140.8
Complex INT ALU	1.84	1.23	9001.72	1.84	1.23	9001.72
Internal rS/External rS	0.22/0.06	0.209/0.01	1115.0/165.19	-	-	-
INT RRB / FP RRB	1.66 / 2.17	11.09 / 16.99	68831.3 / 15565.7	-	-	-
INT WRB / FP WRB	2.1 / 2.27	2.55/ 4.74	7068.6/ 13942.4	-	-	-
I$ vs	-	-	-	0.68	0.19	1455000
Internal rS C$	2.027	0.208	1196851	-	-	-
External rS C$	0.438	0.038	170789	-	-	-
FP/ALUs C$	0.509	0.052	310000	-	-	-
(INT/FP) RRB C$	0.445	0.039	243975	-	-	-
(INT/FP) WRB C$	0.206	0.01	49393	-	-	-
Pre-Load/Ext ARF C$	0.1166	0.01	23845	-	-	-

7.3.3 Cycle Time Determination

Cycle time is derived for the baseline processor by finding the longest (latency) pipeline stage. In this case, cycle time was determined by the FU+bypass pipeline delay for both baseline processors [3], giving a cycle time of 3.5ns for the SMT and STC, where the SMT incurs two additional pipeline stages. For ConSSTEP, cycle time was derived using the following equations:

1) $CycleTime$ = max($T_{exe/engine}$, T_{config}), where
2) $T_{exe/engine}$ = max(rS propagation latency (lat.) + ALU execution lat., RRB lat., WRB lat.)
3) T_{config} = max(read C$ bank lat., write config register lat., read config register lat.)

Table 7.4: Derived cycle time per processor/engine

Processor	**Clock Cycle (ns)**
SMT and STC	3.5
$\langle 2FU \rangle$ Engine	2.2
$\langle 3FU \rangle$ Engine	2.42
$\langle 4FU \rangle$ Engine	2.64
$\langle 6FU \rangle$ Engine	3.0
$\langle 8FU \rangle$ Engine	3.52

$T_{exe/engine}$ considers the worst case latency for traversing data on the rS interconnect from result generation to dependent ALU source input, including any intermediate result latching in the rS. $T_{exe/engine}$ is therefore mainly dependent on propagation latencies and engine size. Conversely, T_{config} is the maximum latency incurred when configuring engine components, considering both the configuration caches and dedicated registers per component. The values for all synthesized latencies are given Table 7.3, and are independent of engine size (considering all latching and combinational delays). Based on these latency derivations, the worst case 4-FU engine cycle time was calculated as follows:

4) $CriticalPath$ = ext rS lat. + int rS lat. + ALU lat.

Therefore, 5x0.22ns+0.06ns+1.48ns = 2.64ns cycle time, where the 6-FU engine

similarly yields a 3ns cycle time etc. This signifies that there is a slight difference in cycle time between engines depending on their configurations, and is strictly dependent on the worst case propagation delay to maintain single-cycle interconnect latency.

Results for all baseline and ConSSTEP engine cycle times are presented in Table 7.4. The SMT operates at the same frequency as the STC, however considering two additional pipeline stages, including thread select and pick. The cycle time of multi-core STC baselines also do not consider interconnect latencies. As seen in the table, in comparison to the conventional CPU, ConSSTEP displayed a significant advantage in terms of achievable frequency due to its simplistic design and scalable data transport. Likewise observing Table 7.3, we observe that FP engines easily achieve the same frequency as integer-based engines. Thus for heterogeneous ConSSTEP configurations which contain multiple engine sizes, **the slowest engine's cycle time determines ConSSTEP's frequency of operation, where the limiting cycle time is applied to all engines**.

7.4 Summary

This chapter outlined the experimental methodology used for deriving and assessing the ConSSTEP core and baseline processors presented in the next two chapters. Specifically, this chapter outlined architectural framework specifications for assessing performance factors, and the details of physical modelling to evaluate factors of latency, power/energy and area overhead across all core types. Now that such evaluation procedures have been clearly stated, the next chapter presents various sensitivity studies to determine optimal scheduling algorithms and topology configurations for the ConSSTEP architecture.

Chapter 8

Sensitivity Studies

8.1 Introduction

This purose of this chapter is to conduct many experiments on the intra- and inter-bundle scheduling algorithms discussed in Chapter 6, assessing the best option(s) for PhysC integration. Specifically factors of performance and energy efficiency must be assessed, in addition to the best combination of the two factors for scheduling efficiency. The algorithms used in this chapter are tested using the multi-threaded SPLASH-2 and PARSEC benchmarks presented in the previous chapter. Accordingly, the specific objectives of this chapter are to:

- Determine the intra-scheduling algorithms which contribute to the least contention and highest performance overall for various ConSSTEP engine configurations. Algorithms with considerable contention will not be considered for PhysC integration.
- Determine the intra-scheduling algorithms which provide low hop count (i.e. lowest rS units traversed) to mitigate data movement, while also considering the performance attained in the previous stage. Thereafter, the best intra-scheduling algorithm may be determined.
- Observe the engine configuration(s) which prove problematic towards performance and/or energy efficiency. Such configurations will not be considered during experimental results which evaluate ConSSTEP to the baseline processor cores.
- Verify the best inter-scheduling algorithm using the intra-scheduling algorithm selected previously. To test for the best algorithm, bundles will be scheduled

using extreme heterogeneous engine topologies.

Based on the results obtained in this chapter, the corresponding best options for the intra- and inter-bundle algorithm will be integrated into the PhysC for the experimental testing conducted in the next chapter. Thereafter during *Experimental Results*, several heterogeneous and homogeneous engine configurations may be tested for various 2-thread (2T) and 4-thread (4T) workloads, and compared to conventional SMT cores and STC multi-core systems.

Accordingly, this chapter will first test the scheduling efficiency of the intra-bundle algorithms Depend (Dep), Dynamic (Dyn), First Come First Serve (FCFS) and Random (Rand), with the inter-bundle algorithms Demand, Static, and Random. Specifically, 2T *homogeneous* engine configurations for ConSSTEP will be assessed across the algorithms, determining how factors of utilization on the interconnect, i.e. rS latch utilization for temporary storage, rS utilization during propagation, and re-routing delays incurred due to utilizations conflicts, affect performance for a given engine topology. Thereafter IPC and hop count are assessed and correlated to the previous utilization and contention results. By solely testing 2T homogeneous cores, each engine's individual behaviour may be understood, where the best engines may be selected for testing and analysis during the experimental results of the next chapter.

8.2 Intra-Scheduling Algorithm Efficiency

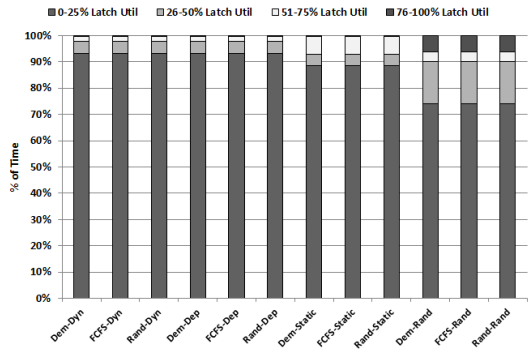
The following section presents experimental testing for all intra- and inter-scheduling algorithms considering 2-Thread (2T) homogeneous ConSSTEP architectures. Engine configurations are listed as $\langle xFUs \text{ x } yFUs \rangle$, signifying dimensions of the engine tested i.e. x FUs on the x-axis, y FUs on the y-axis, for a total of x*y FUs. Engines consisting of 2 or 3 FUs (i.e. asymmetric) with *one y* FU and *multiple x* FUs suggests a *horizontally-oriented* engine configuration (i.e. $\langle$2x1$\rangle$), whereas *one x* and *multiple y* FUs signify *vertically-oriented* engine (i.e. $\langle$1x2$\rangle$). Other engines tested are of symmetric orientations and hence are not referred to as vertically or horizontally oriented. For reasons of asymmetric topological orientation, 5-FU engines have been omitted during experimental testing. Accordingly, configurations with 2FUs, 3FUs, 4FUs, 6FUs, and 8FUs are evaluated during this chapter.

8.2.1 rS Latch Utilization

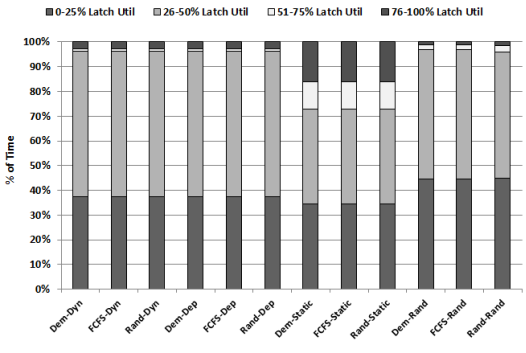
The first factor assessed for ConSSTEP performance and algorithm efficiency is rS storage i.e latch utilization within the rS interconnect. Since an internal register file is not invoked in ConSSTEP, the rS latch usage must be monitored. Consequently, if latches are highly utilized and an insufficient number are available for temporary storage, extra delays are incurred by the engine until a latch becomes available. Latch utilization in this case signifies the number of latches being utilized at a given time. Fig. 8.1 provides the utilization breakdown for each 2T homogeneous configuration, for all scheduling algorithms. Specifically, the graphs outline the percentage of utilized latches for a given amount of time, considering the total duration of the benchmark's execution. Thus in general, the greater the time spent in the higher utilized categories (i.e 76-100%), the more likely the algorithm experiences stalls due to the lack of available latches.

As seen in the figure, the Dep and Dyn intra-scheduling algorithms generally provided the best (lowest) latch utilization, where the Rand algorithm demonstrated slightly lower utilization than Static due to its random placement, effectively reducing storage contention. Comparing the two 2-FU orientations, the ⟨2x1⟩ configuration shown in Fig. 8.1(a) was able to provide lower latch utilization in comparison to ⟨1x2⟩ (Fig. 8.1(b)) due to an increase in propagation contention, discussed in the next subsection. Consequently such propagation contention contributes to more routing delays and likewise lower storage utilization per cycle. Conversely, the ⟨2x2⟩ (4-FU), engine's symmetric orientation, shown in Fig. 8.1(e), was able to provide more storage locations in a distributed manner, with less propagation contention and a wider spectrum of rS utilization. Similar utilization characteristics were also observed in the ⟨3x1⟩ (Fig. 8.1(c)) and ⟨1x3⟩ engine Fig. 8.1(d)) configurations. The vertical orientations in general however were able to distribute latch locations between rows more efficiently across the algorithms, demonstrating higher usage. Therefore as seen in the respective figures, the vertical orientations of the 2- and 3-FU engines (i.e. ⟨1x2⟩ and ⟨1x3⟩) utilized more latches (with less contention impact) than their horizontal counterparts. It was observed that the more rS units integrated on the interconnect in a distributed and spatial manner, the better the performance observed.

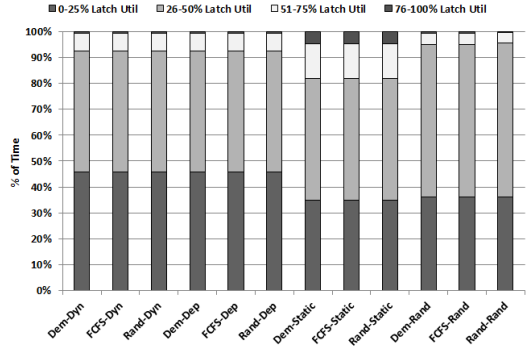
For 4-FU and 6-FU engines shown in Fig. 8.1(e) and (f), respectively, all al-



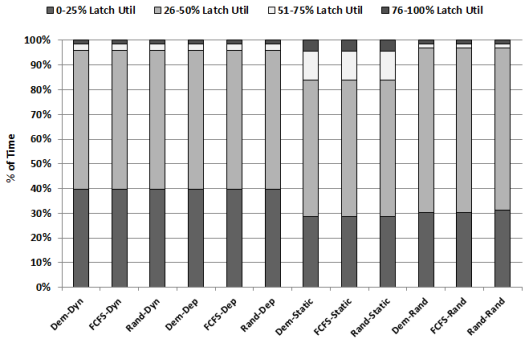
(a) ⟨2x1⟩⟨2x1⟩ Engine (2-FU Engines)



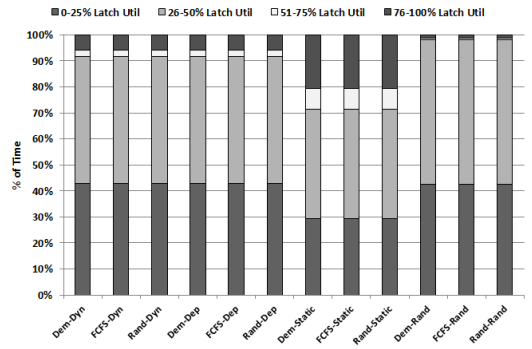
(b) ⟨1x2⟩⟨1x2⟩ Engine (2-FU Engines)



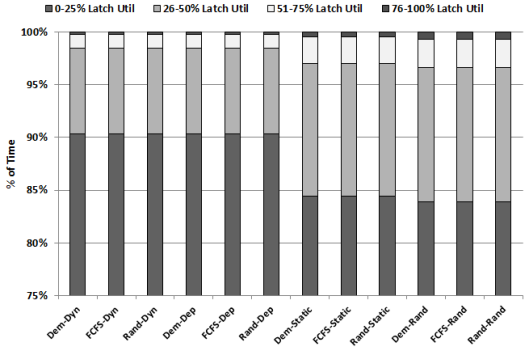
(c) ⟨3x1⟩⟨3x1⟩ Engine (3-FU Engines)



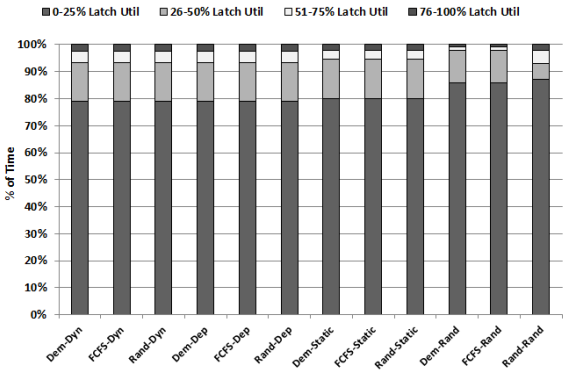
(d) ⟨1x3⟩⟨1x3⟩ Engine (3-FU Engines)



(e) ⟨2x2⟩⟨2x2⟩ Engine (4-FU Engines)



(f) ⟨2x3⟩⟨2x3⟩ Engine (6-FU Engines)



(g) ⟨2x4⟩⟨2x4⟩ Engine (8-FU Engines)

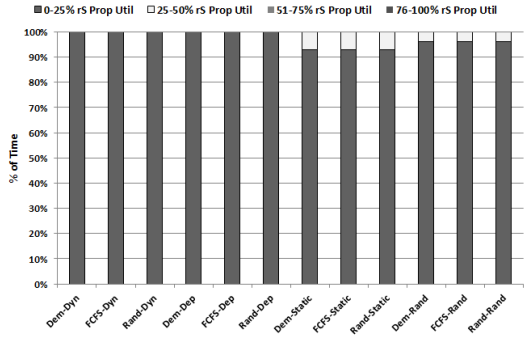
Figure 8.1: rS Latch Utilization Comparison for Engine Configurations

gorithms provided fair utilization as an abundance of latches were present in the network, with minor storage contention issues in comparison to the smaller engines. Accordingly, the Dep and Dyn intra-scheduling algorithms demonstrated the best utilization when compared to Rand and Static. For the 8-FU (Fig. 8.1(g)) engine, the best latch utilization was surprisingly obtained by the Rand algorithm, mainly due to its random nature, which sparsely mapped instructions to allow for low latch utilization. Finally as expected, the *inter*-scheduling algorithms did not effect latch utilization due to the homogeneous architecture tested. Hence no performance gain was attained as both engines provided the same advantages and limitations regardless of the inter-scheduling algorithm.

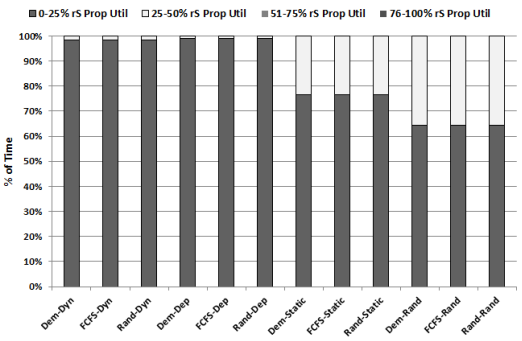
8.2.2 rS Utilization during Propagation

For all homogeneous configurations and algorithms tested, Fig. 8.2 displays rS unit utilization during data propagation with respect to the total duration of a benchmark's execution. As seen in the figure, majority of the applications tested demonstrated a 0-25% utilization of rS units during propagation. For smaller engines such as the 2-FUs of Fig. 8.2(a) and (b), this distribution signifies blocking due to the xy routing protocol. This may be verified in Fig. 8.3 which demonstrates the associated delay incurred during propagation due to contention in the rS interconnect. As verified in Fig. 8.3(a) and (b), contention especially dominates the rS interconnect's lower level for the 2-FU configurations, and hence only a certain number of rS may be utilized per clock cycle.

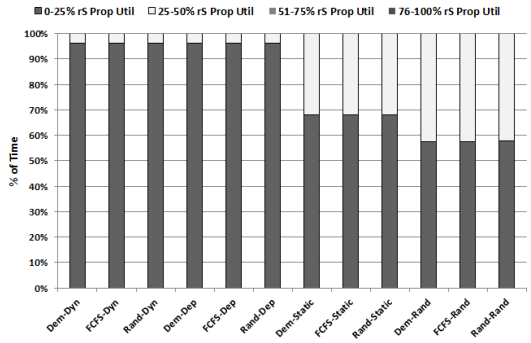
As seen in Fig. 8.3(a) and Fig. 8.3(c), the Static and Rand algorithms generate high contention for such horizontally-oriented topologies, where Dep and Dyn are able to outperform these algorithms due to their intuitive instruction mapping. Considering Dyn and Dep for 4-FU (shown in Fig. 8.3(e)), 6-FU (Fig. 8.3(f)), 8-FU (Fig. 8.3(g)) and $\langle$1x3$\rangle$ (Fig. 8.3(d)), it is seen that these algorithms also utilize the rS interconnect efficiently, which is again due to the scheduling algorithm's ability to directly addressing instruction dependencies and placement. Although not all the rS are utilized in these engines, less re-routing is required in comparison to the other scheduling methods as verified in the respective sub-figures of Fig. 8.3. Overall according to the results obtained, Dep proves best for all configurations types, and especially for SPr benchmarks Barnes and Black (Fig. 8.3) which access memory frequently. Therefore the Dep algorithm is able to adequately reduce propagation



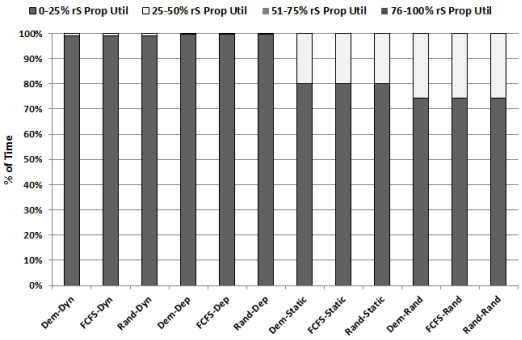
(a) ⟨2x1⟩⟨2x1⟩ Engine (2-FU Engines)



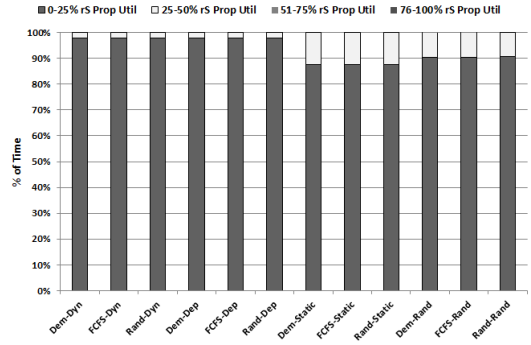
(b) ⟨1x2⟩⟨1x2⟩ Engine (2-FU Engines)



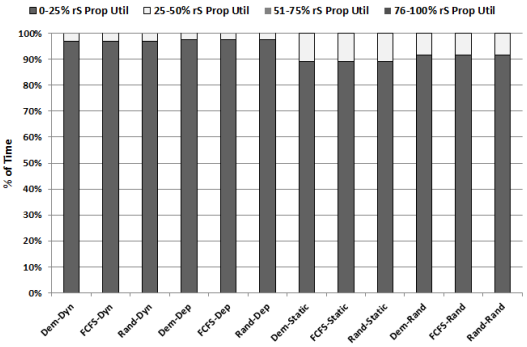
(c) ⟨3x1⟩⟨3x1⟩ Engine (3-FU Engines)



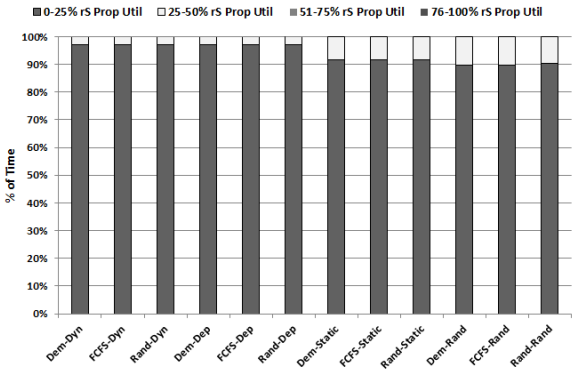
(d) ⟨1x3⟩⟨1x3⟩ Engine (3-FU Engines)



(e) ⟨2x2⟩⟨2x2⟩ Engine (4-FU Engines)

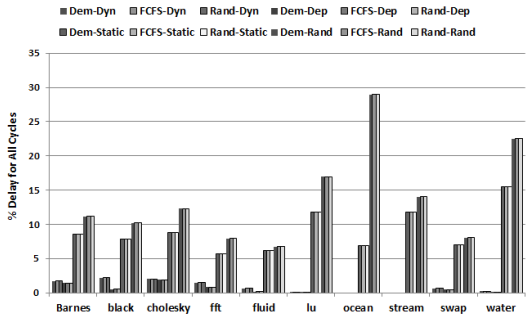


(f) ⟨2x3⟩⟨2x3⟩ Engine (6-FU Engines)

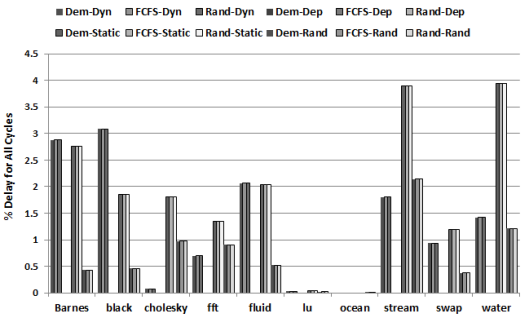


(g) ⟨2x4⟩⟨2x4⟩ Engine (8-FU Engines)

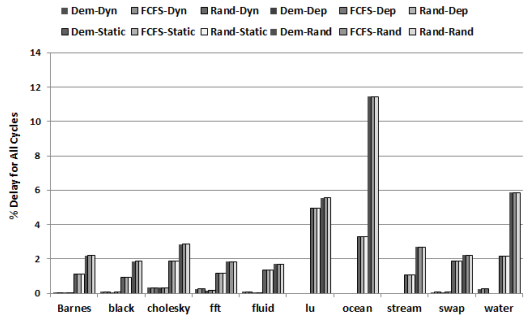
Figure 8.2: Comparison of rS Utilization During Propagation for 2-Thread



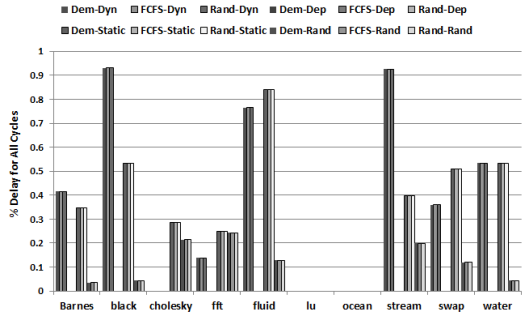
(a) ⟨2x1⟩⟨2x1⟩ Engine (2-FU Engines)



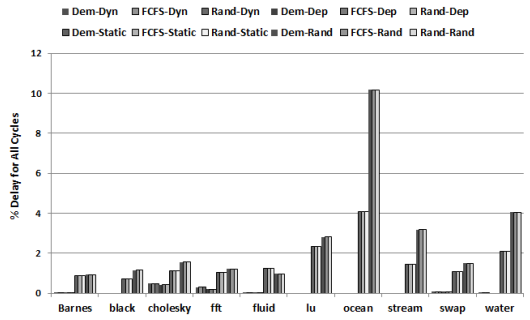
(b) ⟨1x2⟩⟨1x2⟩ Engine (2-FU Engines)



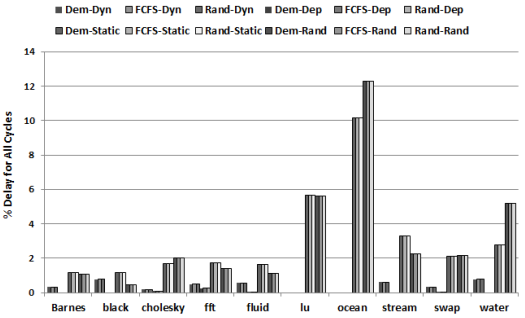
(c) ⟨3x1⟩⟨3x1⟩ Engine (3-FU Engines)



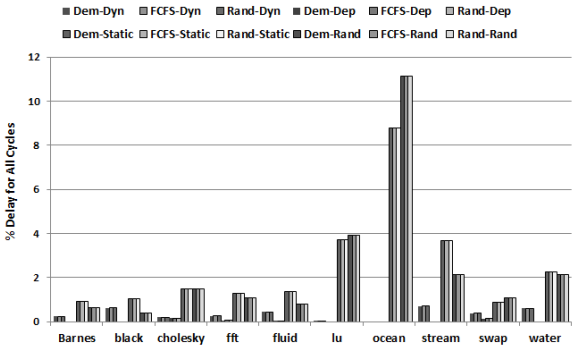
(d) ⟨1x3⟩⟨1x3⟩ Engine (3-FU Engines)



(e) ⟨2x2⟩⟨2x2⟩ Engine (4-FU Engines)



(f) ⟨2x3⟩⟨2x3⟩ Engine (6-FU Engines)



(g) ⟨2x4⟩⟨2x4⟩ Engine (8-FU Engines)

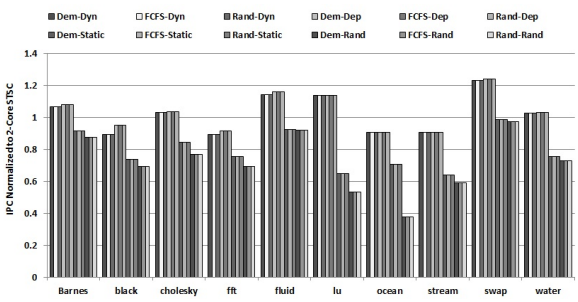
Figure 8.3: Comparison of Delays due to Propagation Contention for 2-Thread

prone contention for such low performing memory-intensive applications. Dyn however does come very close to the Dep algorithm's results, and is especially beneficial for more computationally-intense applications.

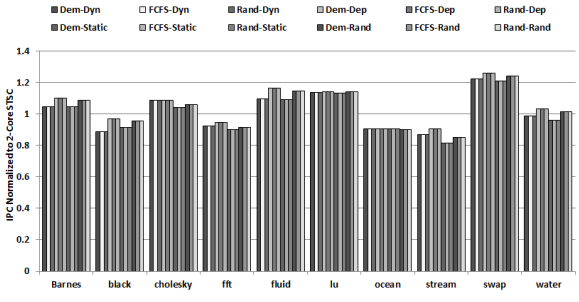
8.2.3 IPC

Fig. 8.4 displays the IPC outcomes for all homogeneous engine configurations and algorithms listed, normalized to the IPC attained by a 2 core STC system, i.e. 1 thread workload/core, average attained IPC, considering a 2T workload. Theoretically, intra-scheduling algorithms scheduled on a homogeneous ConSSTEP architecture should solely impact hop count (the number of Rs units traversed between dependent instructions) and its respective energy efficiency. Thus bundle placement considering a homogeneous setting should not effect IPC. However as discussed in the previous subsections, intra-placement algorithms have a direct impact on both propagation and storage contention, and therefore also impact performance. In addition the results obtained in the previous section, Fig. 8.4 also verifies that the more naive intra-scheduling approaches such as Static and Rand exhibit lower performance (IPC) due to increased contention, causing higher re-scheduling and delay requirements during instruction routing, whereas more involved algorithms such as Dep and Dyn are able to avoid contention. Therefore, Dep again provides the best performance outcome, achieving less rS contention for data stores and propagation of horizontally-oriented configurations. The Dyn algorithm however was also able to achieve performance within 0.8% of the Dep algorithm.

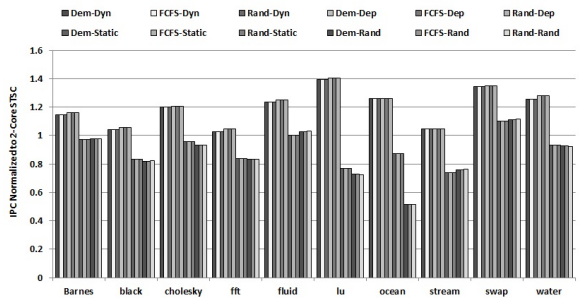
When compared specifically to the Rand and Static intra-placement algorithms, Dep was able to improve performance on average for 2-FU engines by 37.3% (seen in Fig. 8.4(a) and (b)), 3-FU by 18.86% (Fig. 8.4(c) and (d)), 4-FU by 20.63% (Fig. 8.4(e)), 6-FU by 25.74% (Fig. 8.4(f)), and 8-FU by 21.03% (Fig. 8.4(g)). Since more FUs and rS units were available for 8-FU engines, contention was less of a factor across the routing algorithms tested, and hence the 8-FU exhibited more performance improvement than the 6-FU engines in general. Similarly, the $\langle 1x3 \rangle \langle 1x3 \rangle$ ConSSTEP architecture received very little performance variation across all algorithms as seen in Fig. 8.4(d). However referring back to the contention displayed in Fig. 8.3(d), it is evident that contention in general was also very minimal due to the vertical topology layout, and therefore negligible variation in IPC.



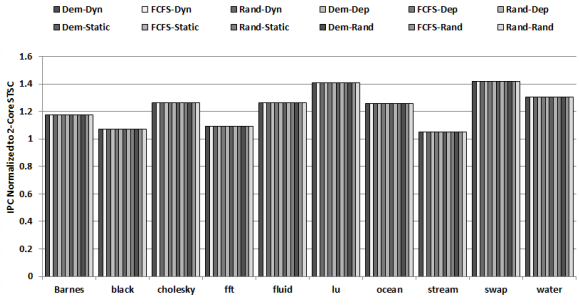
(a) ⟨2x1⟩⟨2x1⟩ Engine (2-FU Engines)



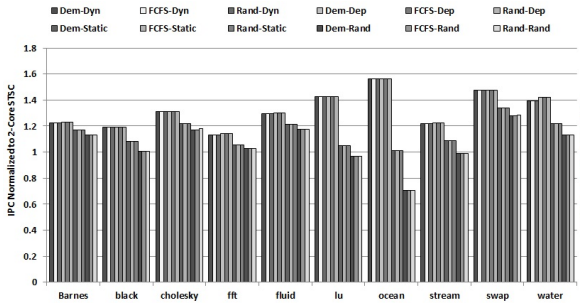
(b) ⟨1x2⟩⟨1x2⟩ Engine (2-FU Engines)



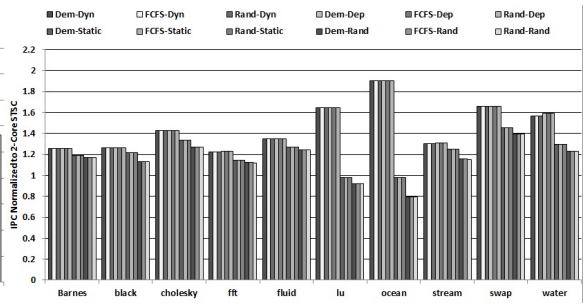
(c) ⟨3x1⟩⟨3x1⟩ Engine (3-FU Engines)



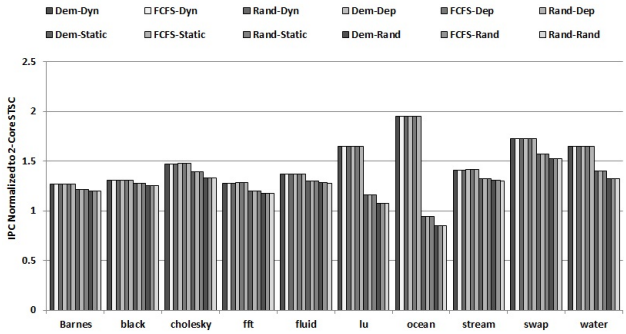
(d) ⟨1x3⟩⟨1x3⟩ Engine (3-FU Engines)



(e) ⟨2x2⟩⟨2x2⟩ Engine (4-FU Engines)



(f) ⟨2x3⟩⟨2x3⟩ Engine (6-FU Engines)



(g) ⟨2x4⟩⟨2x4⟩ Engine (8-FU Engines)

Figure 8.4: IPC - Performance Comparison for 2-Thread Engine Configurations

The most important observation however was that the 2-FU engines were not able to exceed the IPC of a conventional STC, defeating the purpose of a ConSSTEP core. It is also evident that the vertically-oriented engine topologies provides better performance and generally less contention than their horizontal counterpart.

8.2.4 Hop Count

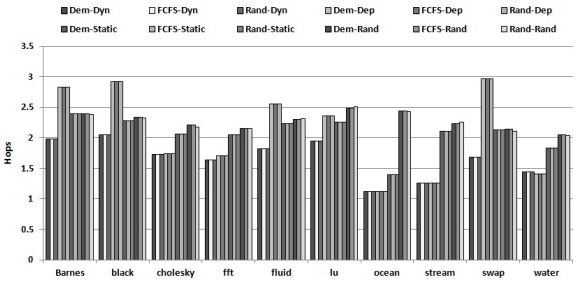
Fig. 8.5 provides the hop count distribution for the scheduling algorithms tested across all benchmarks. Hop count is directly associated with the energy efficiency of data movement i.e. the less hops/rS units traversed, the more energy efficient the algorithm. Therefore the most ideal intra-scheduling algorithm will provide both high performance and low hop count.

As seen in Fig. 8.5, the Dep and Dyn intra-bundle scheduling algorithms were able to provide the best hop count, improving upon the other algorithms in the range of 8-28%. According to the figure, the horizontally-oriented configurations provide lower hop count and therefore higher energy efficiency when compared to the vertically-oriented engines for the 2 and 3-FU topologies. As seen in the previous subsection however, such hop count mitigation was at the expense of some performance loss in these configurations.

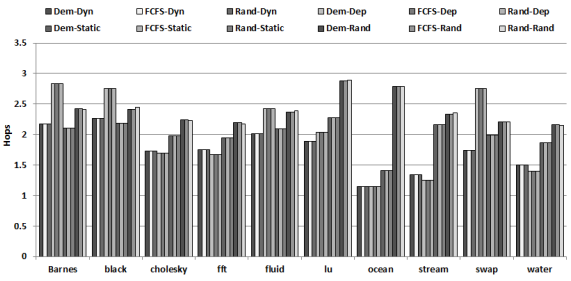
Observing the results between the two most efficient algorithms, Dyn provides approximately 22.8% more hop efficiency than Dep. That is, since Dep places greater priority on the data traversal of external operations, dependencies of other instruction located in the main grid are given second priority, in turn increasing hop count in the main engine. Therefore, such results demonstrate that focusing on main grid dependencies proves beneficial to energy efficiency, and especially for the more computationally-intensive workloads. Therefore the Dyn intra-bundle scheduling algorithm provides the best solution for reducing hop count, and improving energy efficiency.

8.2.5 Summary - Intra-Scheduling

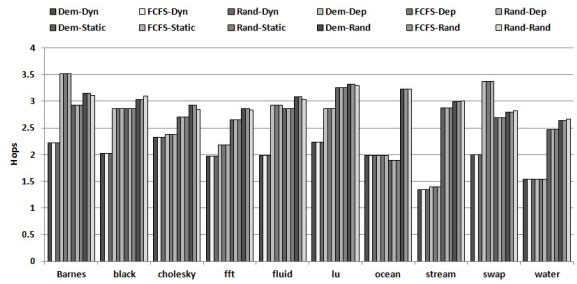
Fig. 8.6 and 8.7 display the average performance and hop increase, respectively, considering a 2T architecture per intra-scheduling algorithm, normalized to a 2-core STC for all benchmarks. Since intra-scheduling algorithms gave the same outcomes



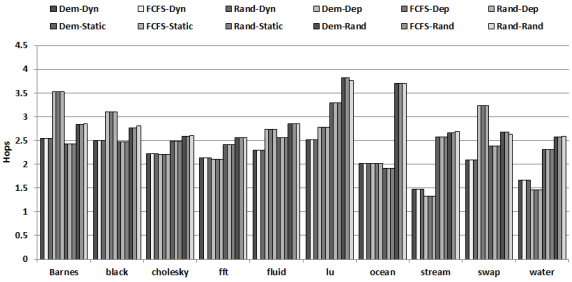
(a) ⟨2x1⟩⟨2x1⟩ Engine (2-FU Engines)



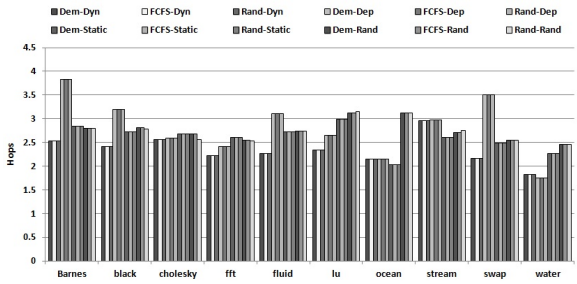
(b) ⟨1x2⟩⟨1x2⟩ Engine (2-FU Engines)



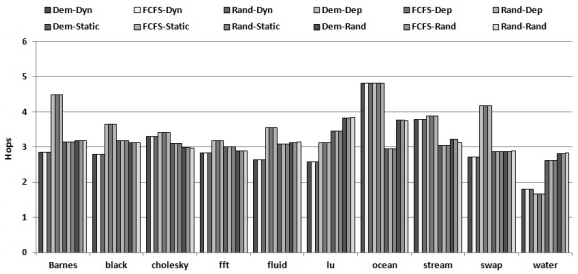
(c) ⟨3x1⟩⟨3x1⟩ Engine (3-FU Engines)



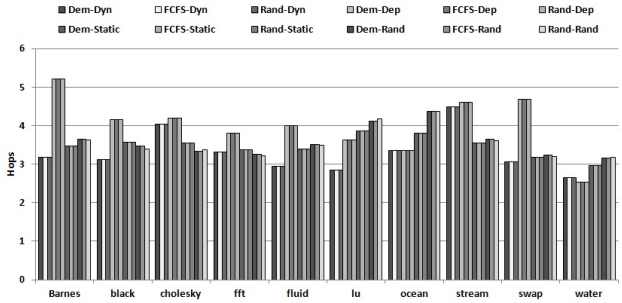
(d) ⟨1x3⟩⟨1x3⟩ Engine (3-FU Engines)



(e) ⟨2x2⟩⟨2x2⟩ Engine (4-FU Engines)



(f) ⟨2x3⟩⟨2x3⟩ Engine (6-FU Engines)



(g) ⟨2x4⟩⟨2x4⟩ Engine (8-FU Engines)

Figure 8.5: Hop Count Comparison for 2-Thread Engine Configurations

for all inter-scheduling algorithms, only intra-algorithms are provided in these figures. As seen in Fig. 8.6, the greater the number of FUs present in an engine, the more likely the PhysC was able to take advantage of the wider issue width, which also mitigates contention to provide further performance gains. Also observed in the figures are the trends of the vertically-oriented topologies which achieve slight performance advantages to their horizontal counterparts, mainly due to reduced contention and more distributed storage locations.

In terms of energy efficiency and hop count as displayed in Fig. 8.7, it is observed that the Dyn intra-scheduling algorithm provided the best efficiency, where Static was able to match Dep hop count in certain cases. Considering both performance and energy efficiency however, the *Dyn intra-scheduling* algorithm proves most efficient, and will therefore be used for further experimental testing in the subsequent sections.

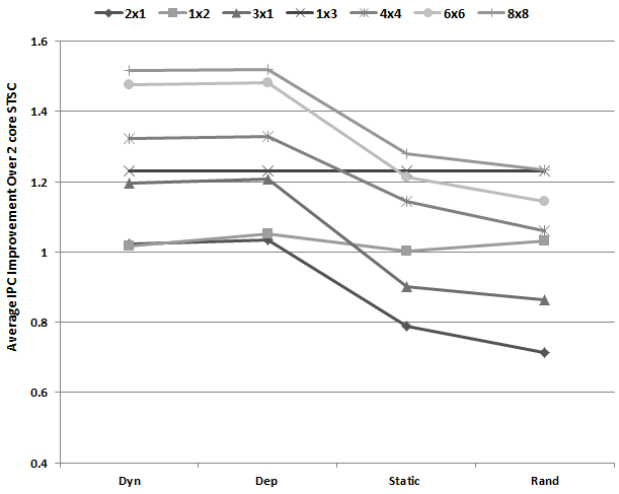


Figure 8.6: Average IPC per Algorithm for all Benchmarks - 2T Workload

As previously mentioned, this thesis also deduces by analysis that both 2-FU engine orientation were problematic for achieving performance gains, and providing adequate storage area for temporary values. Specifically, performance is lower than that of a STC, defeating the purpose of such an architecture for high single-thread and throughput performance. Thus all 3-FU engines and larger will be considered during experimental testing for evaluating ConSSTEP's feasibility and advantage over multicore STC and SMT processors.

The subsequent section will present an extreme heterogeneous configuration case, demonstrating the performance effects of inter-scheduling algorithms in a brief and

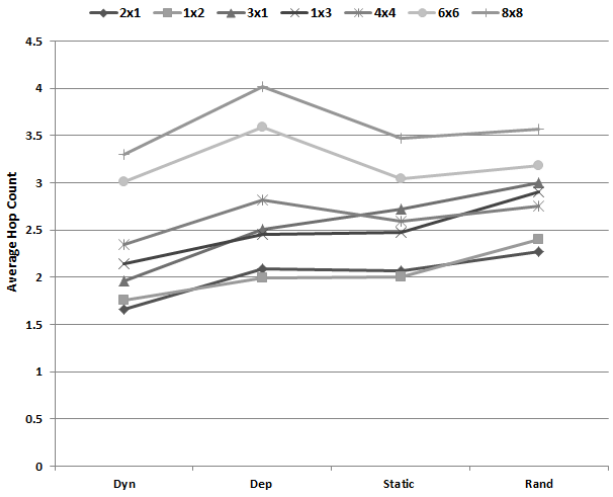


Figure 8.7: Average Hops per Algorithm for all Benchmarks - 2T Workload

concise manner, as we have clearly observed in this section that more involved algorithms are optimal for routing and storage placement problems. Thus the inter-scheduling algorithms which map thread bundles to engines i.e. Demand, FCFS and Rand will be tested using a 4-threaded core with extreme variation in engine sizes, using ⟨1x2⟩ (2-FU) and ⟨2x4⟩ (8-FU) engines, while using the Dyn intra-scheduling algorithm to map instructions to their respective FUs. Since there is much variation in the engine sizes selected, the mappings and respective performance outcomes will also quickly demonstrate the performance effects of inter-bundle placement. Based on the outcome, the Experimental Results section will then employ the best algorithm pair, invoking various engine configurations to find the most efficient and suitable ConSSTEP architecture for single-thread and throughput performance on a single core.

8.3 Inter-Scheduling: Bundle to Engine Mapping

Figure 8.8 presents the results of a ⟨1x2⟩⟨2x4⟩⟨1x2⟩⟨2x4⟩ 4-Threaded (4T) ConSSTEP architecture, comparing the Demand, Random, and FCFS inter-algorithms, paired with the Dyn intra-scheduling algorithm. As seen in the Figure, the Demand algorithm surpasses the FCFS algorithm by 1.11x, whereas Demand observes a 1.152x speedup in comparison to Rand. Thus the Depend algorithm (as expected) outperforms and/or matches in the worst case the performance of the other inter-scheduling algorithms, considering a 4-Thread processor. Accordingly, the next section will present various 2T and 4T ConSSTEP configurations considering the

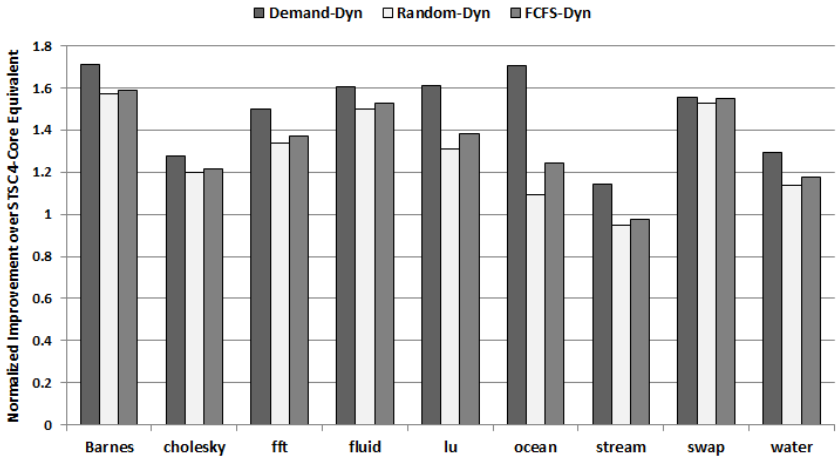


Figure 8.8: Heterogeneous Performance Analysis of Inter-Scheduling Algorithm - 4-Threads

Demand-Dynamic (Dem-Dyn) algorithm, while using the engine combinations of (vertically-oriented) 3-FU, 4-FU, 6-FU and 8-FU engines.

8.4 Summary

This chapter provided various sensitivity studies to determine the ideal ConSSTEP scheduling algorithms for the PhysC to attain the best overall performance and energy efficiency. Based on results, it was determined that the Demand-Dynamic (Dem-Dyn) algorithm combination generally provided the best results for all engine combinations. Based on the tests, it was also determined that the 2-FU engine configurations, both vertical and horizontal, suffered performance losses largely due to contention for the topological orientation of the core and lack of storage locations. Therefore the next chapter will compare various 3-FU (vertically-oriented), 4-FU, 6-FU and 8-FU engine configurations to the baseline cores, determining ConSSTEP's feasibility as a configurable multi-threaded processor.

Chapter 9

Experimental Results and Analysis

9.1 Introduction

The previous chapter determined the most efficient scheduling algorithm for the PhysC in terms of performance and energy efficiency - Dem-Dyn. This chapter integrates the algorithm into the PhysC, and assesses various engine configurations according to the topologies verified in the previous chapter, comparing such variations of ConSSTEP cores to conventional SMT (multi-core) single-thread core (STC) baselines. Such comparisons are used to determine factors of performance, energy efficiency, and area overhead which may validate the feasibility of ConSSTEP. The most ideal ConSSTEP engine configurations will also be determined for 2T and 4T workloads. Accordingly, this section will assess the following factors and answer the following questions:

- **Performance**:
  - **Single-thread IPC** improvement: One of ConSSTEP's main objectives is to increase single-thread performance for multi-threaded workloads. Based on the results provided in this chapter, the following chapter will determine whether ConSSTEP was able to provide an improvement in single-thread IPC in comparison to SMT. And more importantly, was ConSSTEP able to achieve a single-thread performance gain in comparison to a STC?
  - **Throughput** improvement. Throughput considers attainable frequency and how fast an application may execute in instructions/second. Thus the topology of an engine and its simple logic may allow ConSSTEP to

execute at a faster rate than complex conventional CPUs. Is ConSSTEP therefore able to provide similar or better throughput when directly compared to a SMT considering all benchmarks? Does ConSSTEP provide better throughput than a SMT provides over a STC?

- **Power and Energy:** Power is expressed in Watts (W) representing the instantaneous rate of energy transfer for both the active (dynamic) and inactive (static) transistors/components on a core. Energy conversely is measured in Joules (J), i.e. Watts per second, and expresses the power required of a core within a specified time frame (i.e. switching energy). Energy in this chapter therefore signifies the number of joules required to complete an operation and in this case the energy required to execute a given application. Likewise, an increase in frequency may possibly lead to less energy per application due to such execution speedup, however may also contribute to an increase in dynamic power consumption. Thus both power and energy are taken into consideration to assess and compare ConSSTEP to the baselines. This section of the chapter will then answer the following questions:

  – Does ConSSTEP increase energy efficiency for multi-threaded applications in comparison to a SMT core?

  – As ConSSTEP scales across various core configurations, are the number of FUs required per engine, its frequency, and its respective attainable performance justified in comparison to a SMT core when considering static and dynamic power consumption?

  Only an SMT will be compared in this section to assess power and energy as a multi-core STC will consume much more than ConSSTEP cores (i.e. a core replicated many times on a die).

- **Area overhead:** Considering the performance and energy efficiency of ConSSTEP, what is the relative die size in comparison to the baselines to achieve such gains? Was ConSSTEP able to simplify pipeline logic and reduce die overhead even though it relies on the replication of engines for single- and multi-thread execution efficiency?

- **Performance/unit area** and **Performance/Watt:** Based on the parameters obtained above, the rate of computation per unit area, and the rate of computation per watt may be scaled and compared across the cores. That

is, the performance results may be directly compared considering a fixed area and power constraint per core for both ConSSTEP and the SMT core. Such metrics are especially important in parallel computers as the cost of powering a processor may outweigh the cost of the CPU itself [75].

Using the experimental inquiries discussed above, the results presented in this chapter will determine the feasibility of ConSSTEP on various levels in comparison to a conventional SMT and multi-core system. As previously discussed, 2T and 4T configurations are assessed for all core types where ConSSTEP is constrained by the maximum size of an equivalent SMT die. Specifically, the first section of this chapter addresses 2-thread (2T) processor prototypes for performance, power/energy and area, where the findings are then extended to 4-Thread (4T) prototypes in the second section of the chapter. Finally, the chapter then addresses other architectural details and results, including configuration memory requirements and general statistics of propagation and storage through coarse-grained execution by revisiting the study conducted by Tseng and Patt [6].

9.2 Two-Thread Comparison

9.2.1 Configuration Overhead Concealment Techniques

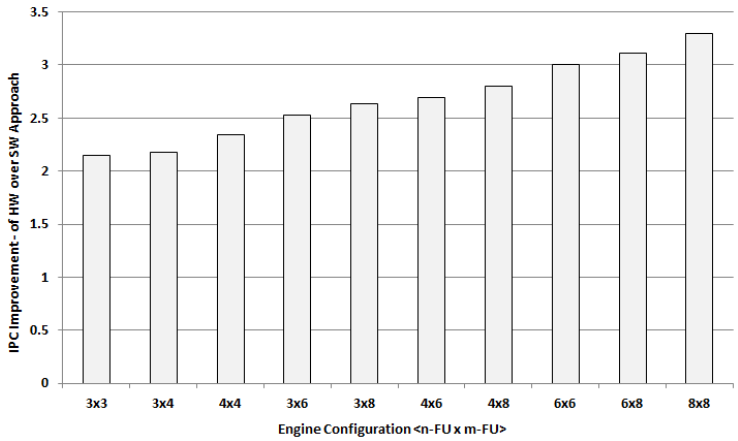


Figure 9.1: Configuration Overhead Performance - Hardware Double Configuration Register Improvement over Software PhysC Scheduling Technique

Although the sensitivity studies provided in the previous chapter considered a hardware approach using a double configuration technique, this section provides

insight on both a software and hardware technique for concealing overhead. Specifically, this section presents the general performance attained by ConSSTEP using two overhead concealing techniques discussed in chapter: 1) Software: the PhysC's determination of the maximum clock cycles needed to configure a given engine per thread, versus 2) Hardware: a double configuration register approach, where each configuration register is pre-configured with the predicted bundle determined by the scheduler. The engine configurations presented in this chapter are expressed as $\langle n$-*FU* engine x m-*FU* engine$\rangle$ using the $\langle x$ x $y\rangle$ topology annotations derived in the previous chapter, where the number of FUs per engine equals x*y and the total FUs per core equals n+m.

Fig. 9.1 presents the IPC improvement which the hardware approach provides over the software technique, averaged across all benchmarks. As seen in the figure, the hardware approach for all 2T configurations provided more than a 2x performance gain as the method purely mitigated overhead penalty except in the case of mispredictions, whereas a PhysC technique was only able to conceal a few clock cycles of the total 10 c.c penalty per configuration. As also observed in the figure, the larger the engine, the greater the performance gain using the hardware approach since the larger engines are generally able to increase issue width and performance. Therefore such aggressive execution with overhead concealment provides more of a performance gain for such engines.

As verified by the results obtained here, the double configuration register approach achieves the best performance and will continue to be used for the experimental testing of this work.

9.2.2 Area

The area of ConSSTEP and the baseline SMT ARM CPU for 2-Threads (2T) are modelled according to the specifications previously discussed in Chapter 7 and provided in Table 7.1. Area comparisons for the several ConSSTEP configurations and baseline SMT CPU models are provided in Table 9.1, presented in mm^2. As seen in the table, the baseline for a 2T ARM CPU possesses the highest area overhead overall. The ConSSTEP core containing 3 FUs - $\langle$3x3$\rangle$ (i.e. $\langle$1x3$\rangle\langle$1x3$\rangle$) was able to decrease area by 56.3%, whereas the largest engine consisting of 8-FUs still de-

Table 9.1: 2-Thread Area Comparison (mm^2)

Processor	**Area (mm^2)**
2-Thread SMT	15.87
⟨3x3⟩	6.928
⟨3x4⟩	6.97
⟨4x4⟩	6.94
⟨3x6⟩	6.987
⟨4x6⟩	6.985
⟨6x6⟩	7.485
⟨3x8⟩	7.048
⟨4x8⟩	7.475
⟨6x8⟩	7.51
⟨8x8⟩	7.523

creased area by 52.6% in comparison to the SMT. Similarly, as seen in the figure the 4-FU and 6-FU engines also decreased area overhead for 2T cores.

When assessing the transistor budget for all processor structures, ConSSTEP dedicated approximately 5-10% of its die area for computing, 5-17% of the die to scheduling, and 73-90% for configuration memory depending on the engine size. This hardware allocation was also considered at a smaller die size in comparison to the SMT. Similarly, the SMT CPU die dedicated roughly 80% of its die area to instruction memory, only 4-5% to computation, and the remaining 15-16% to hardware which rediscovers and eliminates data dependencies and maintains program order - all of which are known by the compiler and imposed by the ISA. Thus ConSSTEP was able to alleviate such unnecessary hardware by invoking the PhysC and redesigning the datapath to save area overhead per die, however allotting much more memory for configuration purposes.

9.2.3 Energy and Power

Fig. 9.2 demonstrates various 2T ConSSTEP cores and their respective energy reduction for all benchmark applications in comparison to a 2T SMT processor. The figure only considers the energy required of the active structures to execute the given

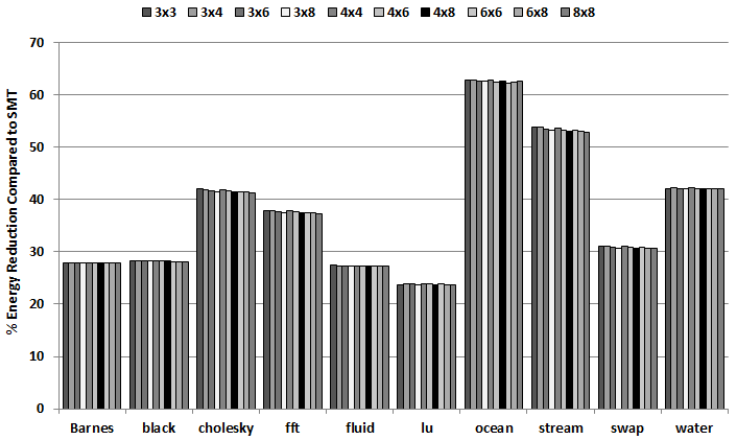


Figure 9.2: Energy Reduction for Various ConSSTEP Configurations

application, referred to as switching energy. As seen in the figure, the 2T configurations are able to save on average 37.5% energy, with the ⟨3x3⟩ engine obtaining slightly more energy efficiency (37.68% on average). Energy consumption differs between configurations in the magnitude of pJ - nJ since only a few more links and rS units may be active for benchmark execution, acknowledging that only the switching energy required to execute the application is calculated. Hence values plotted per benchmark demonstrated very little energy variation between configurations when directly compared to the SMT, however still significant when compared to its 2T SMT equivalent.

As previously mentioned, energy reduction is attributed to the speedup in execution time (performance further discussed in the next subsection) and therefore for certain applications, larger engines may demonstrate higher energy efficiency as they are able to extract higher ILP, generally executing applications faster and requiring less energy. As exhibited in the figure however, ⟨3x3⟩ is able to operate at a higher frequency and achieves the best energy efficiency across all benchmarks. However for the more memory-intensive and SPr applications such as Barnes and Black which are bound by the memory system, engine size contributes very slightly to energy efficiency as speedup is still bounded by load/store instructions. In these cases, larger engines are actually less efficient as the applications may activate more components overall with little speedup in comparison to the other engine configurations.

Fig. 9.3 presents the energy distribution per component for all applications, averaged across all 2T ConSSTEP configurations, as all configurations demonstrated

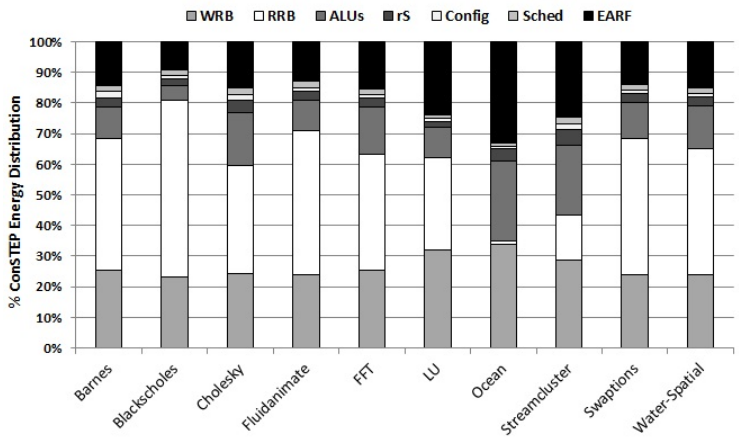


Figure 9.3: Energy Distribution for ConSSTEP Structures, Averaged for all Benchmarks

roughly the same relative energy consumption per component considering identical computation and external read/write requirements per benchmark. As observed in the figure, energy distribution was highly reliant on the application's characteristics. For instance, applications such as Ocean with high arithmetic intensity and fewer memory and branch instructions, demonstrated FUs (integer and FP) accounting for a bigger share of the energy consumption in comparison to the other benchmarks. Although LU and Water were also computationally intensive, they experienced higher shared write accesses and higher memory accesses, respectively, and as observed in the figure, a larger portion of energy was dedicated to the RRB for interfacing external units. Benchmarks such as Blackscholes which experienced higher banking conflicts (i.e. SPr workloads) and branch instructions also required frequent access to external units as illustrated in Fig. 9.3. Therefore higher energy consumption was observed for the WRB and RRB structures for such memory demands. A direct correlation between the EARF energy and the number of external read/writes required by application bundles was also observed, i.e. Ocean and Stream, generally required more EARF reads/writes per bundle than the average bundle. The energy dissipation for configuration (Config) which included mispredicted bundles, accounted for a small portion of the overall energy consumption and therefore configuration was a minimal energy penalty for the ConSSTEP core. Similarly, rS energy accounted for roughly 3-4% of the total energy due to its simplistic design and the PhysC's efficient hop count algorithm.

Next, Fig. 9.4 displays the energy distribution for the 2T and 4T SMT pro-

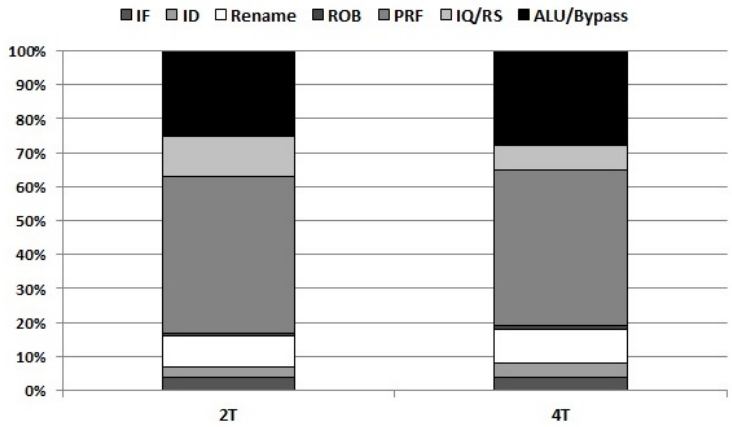


Figure 9.4: SMT Energy Distribution for 2T and 4T SMT

cessor cores. According to the results observed, approximately 80-83% of the SMT energy consumption was attributed to the backend, including reads and writes to the register file, IQ, and bypass network communication. Observing Fig. 9.3, ConSSTEP in comparison eliminated such requirements by 20-30% on average through configuration logic and the rS interconnect, propagating values only when necessary and maintaining scalability without conventional CPU impositions and structures. Accordingly, the rS interconnect mitigated register file energy consumption on average by 9.71x and 7.77x for the 2T and 4T SMT respectively. In turn, these savings amounted to an average of 52.37% less register reads and 55.43% register writes+bypass for all applications. Similarly, the elimination of conventional front-end structures roughly accounted for 15% energy savings of the SMT's total energy, in addition to the ROB and IQ which attributed to 20% of the overall SMT energy.

Fig. 9.5 displays the data movement energy required per engine, averaged across the benchmark kernels. According to the figure, ⟨3x3⟩, ⟨3x4⟩ and ⟨4x4⟩ provided the most efficient data movement for the ConSSTEP processor, saving approximately 13% energy compared to the other configurations. As expected, the most energy consuming were the larger engines, ⟨8x8⟩ and ⟨6x8⟩, consuming up to 41% more energy in comparison to the smaller engines. However referring back to Fig. 9.2, such data movement energy consumption is minimal when directly comparing energy savings to the SMT core, yet this factor must still be considered as transistors scale.

Finally, Fig. 9.6 presents the power savings (dynamic and static) of the various

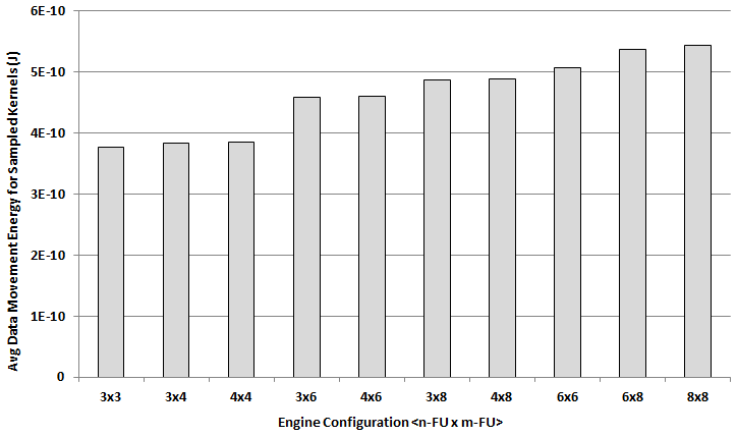


Figure 9.5: Comparison of Data Movement Energy for 2T ConSSTEP Configurations

2T ConSSTEP configuration, normalized to a 2T SMT core and averaged for all benchmarks. As expected, ⟨3x3⟩ was the most efficient core, saving approximately 59% power when compared to the SMT due to its simplistic logic and design. Conversely, the ⟨8x8⟩ configuration saved the least power (32%), with an average savings of 47% across all 2T configurations. As seen in the figure, as more functional units are added to the core, power savings are decreased due to both static and dynamic power consumption of the structures for a given ConSSTEP core.

9.2.4 Performance

Instructions Per Cycle (IPC)

Fig. 9.7 presents the *single-thread* IPC improvement for all baseline models, normalized to a 2T SMT core. The first column in the figure presents the improvement which an average core on a multi-core STC system provides over a 2T SMT core. As displayed, ConSSTEP configurations on average provide an improvement of 1.39x, where the 2-core STC achieves a 1.25x single-thread IPC improvement in comparison to its SMT equivalent. Accordingly, ConSSTEP also improves the single-thread performance of a STC on average by 1.11x across all benchmarks. As seen in the figure however, certain IPC fluctuations exist across engine configurations, in particular the performance attained by heterogeneous engines in comparison to homogeneous core configurations (for example ⟨3x8⟩ versus ⟨4x4⟩). Since engines may only be assigned to one thread's workload, the more algorithmic intensive bundles which exist in a thread (considering a heterogeneous core) may be mapped to a smaller engine

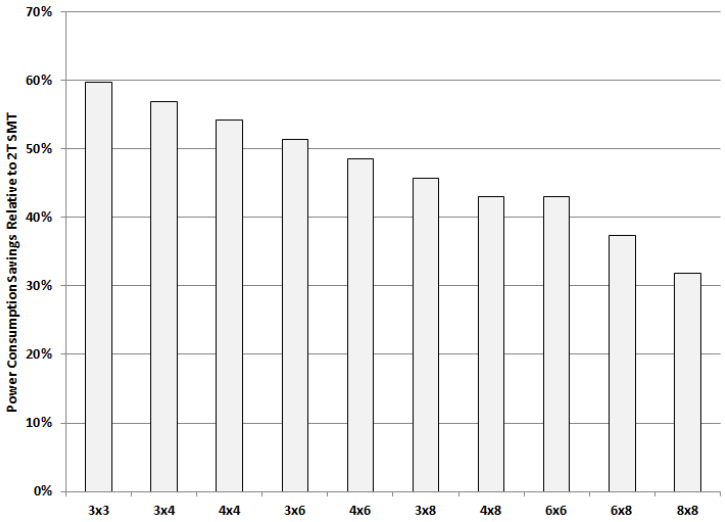


Figure 9.6: Power Consumptions Savings for 2T ConSSTEP Configurations, Normalized to SMT

versus the ideal larger engine. Hence minor performance loss may be exhibited due to the PhysC's thread mapping algorithm within heterogeneous cores. Despite these effects, ConSSTEP's performance gain is still substantial when compared to conventional SMT cores and STC models.

Referring to the individual benchmarks of Fig. 9.7, it is evident that Ocean obtained the greatest single-thread IPC improvement due to its high computational intensity and inherent ILP. Since thread sharing and blocking in the Ocean benchmark was exhibited in the SMT pipeline, larger ConSSTEP configurations were able to adequately support such demanding thread workloads. Conversely, benchmarks such as FFT (which experienced high banking conflicts for independent sequential memory accesses) and SPr workloads (Barnes and Blackscholes) received a small margin of performance improvement with 2T ConSSTEP (approx. 25%). Specifically, benchmarks such as Blackscholes and FFT exhibited less performance gains for the smaller engines due to memory scheduling conflicts across the threads. However the larger engines which possess larger issue widths were able to overlap more instruction execution to mitigate such memory conflicts in comparison to the smaller engines which stalled more frequently and therefore achieved similar results to the SMT. Such benchmarks however with more memory-intensive applications and/or conflicting accesses in general achieved similar results to the standard 2T SMT, however with slight single-thread IPC improvement of approximately 1.77% or greater

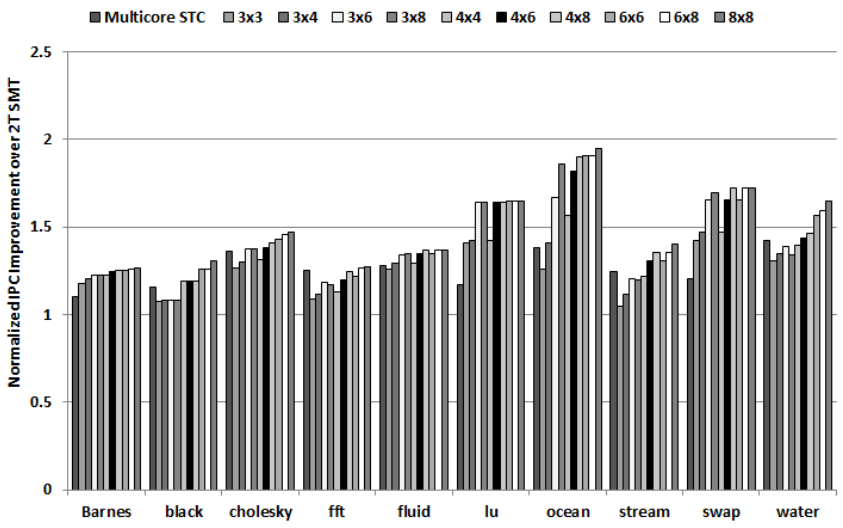


Figure 9.7: Single-Thread IPC Improvement Over 2-Core STSC

in comparison due to increased frequency.

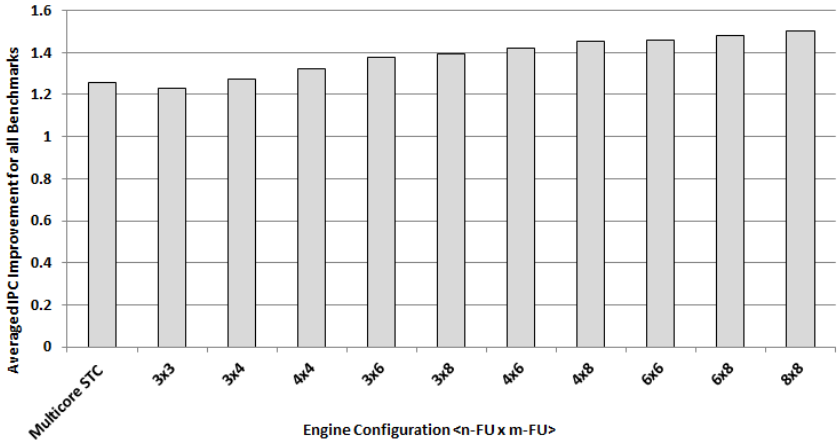


Figure 9.8: Single-Thread IPC Improvement per 2T Configuration, Averaged for all Benchmarks (Normalized to 2-Core STSC)

In accordance with Fig. 9.7, Fig. 9.8 displays the *average* single-thread IPC improvement for the 2T ConSSTEP configurations, averaged across all benchmarks and normalized to the 2T SMT. According to the results obtained, ⟨3x3⟩, ⟨3x4⟩ provide roughly the IPC same improvement to an average STC, whereas the other engines were able to support wider issue widths which benefited certain benchmarks as discussed previously to provide higher IPC gains.

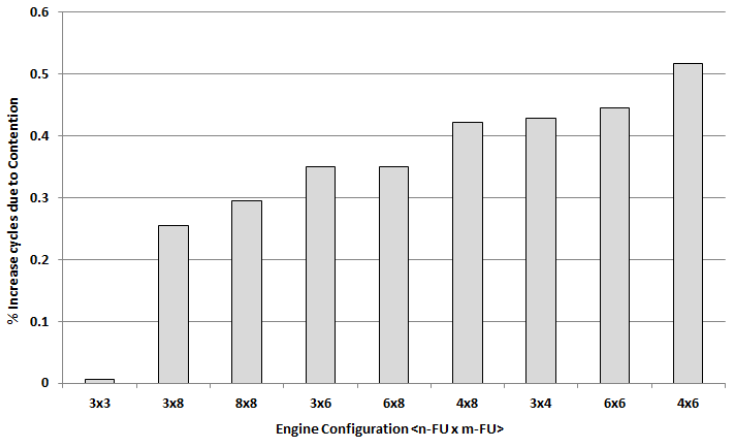


Figure 9.9: Cycle Increase Due to Contention for Various ConSSTEP Configurations

Fig. 9.9 demonstrates the percent increase in cycle required by a given configuration due to contention exhibited by the Demand-Dynamic algorithm across all benchmarks (determined to be the most efficient for both performance and energy). As seen in the figure, all configurations contributed to less than a 0.6% increase in cycles due to contention. Particularly interesting is ⟨3x3⟩ (i.e. ⟨1x3⟩⟨1x3⟩ topology) where the PhysC is able to comfortably re-route and prevent majority of contention due to its vertical topological orientation.

Throughput

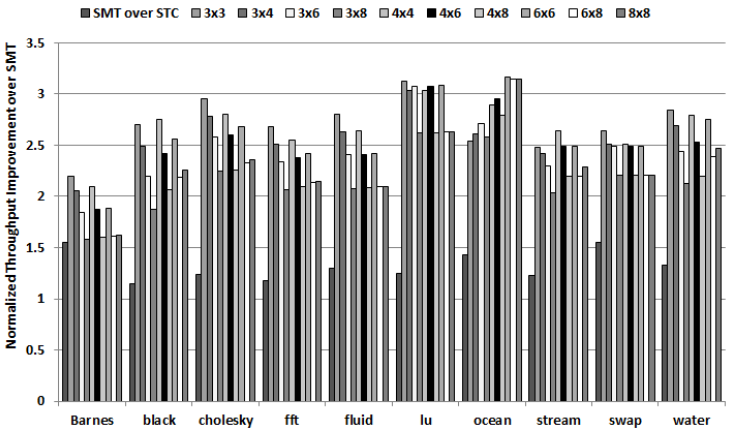


Figure 9.10: Two-Thread Throughput Improvement Normalized over SMT

Fig. 9.10 and 9.11 demonstrate the throughput per benchmark, and average throughput for all benchmarks, respectively, for all 2T engine configurations. The

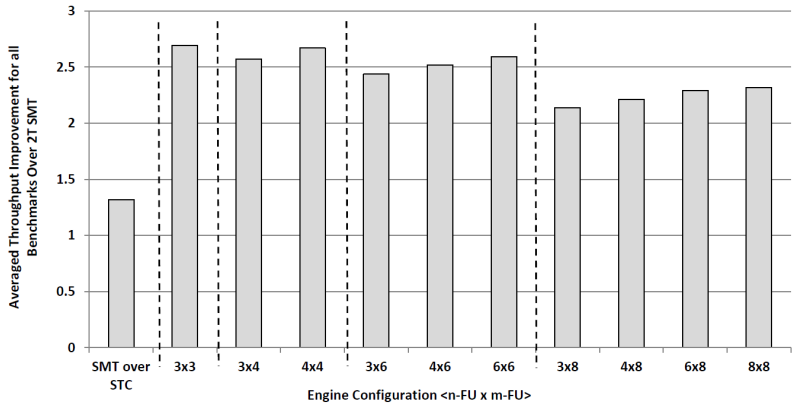


Figure 9.11: Averaged 2T Throughput Improvement Normalized over SMT

first column in each graph illustrates the SMT improvement over an average single-thread core within a dual-core system. Contrary to IPC improvements presented in the previous section, throughput also considers the frequency of operation per engine, where ConSSTEP's frequency is **limited by the slowest engine present on the core** which is applied to all engines. Accordingly the dashed lines of Fig. 9.11 are used to categorize configurations according to the slowest engine presenton a core. As seen in the figures, smaller engines present in more extreme configuration cases such as ⟨3x8⟩ suffer performance losses as the ⟨3⟩ engine's primary advantage is running at a higher frequency versus exploiting issue width/ILP as the case of the ⟨8⟩ engines. Therefore bundles mapped to the ⟨3⟩ engine suffer a performance loss due to such a design limitation. As such, cores with 3-FU engines benefit when paired with other smaller engines. Fig. 9.11 also demonstrates that homogeneous ConSSTEP configurations generally outperform the heterogeneous configurations within the same frequency category, as each engines within the homogeneous core is able to operate at the maximum issue-width and frequency for the given classification.

Fig. 9.10 and 9.11 demonstrate that ConSSTEP configurations achieve a performance advantage over both SMT and STC multi-cores. According to the results obtained, the SMT on average achieves 1.32x the throughput of a single core, whereas the ConSSTEP engines achieves 2.4x the throughput of a SMT core due to frequency scaling. As seen in Fig. 9.10, benchmarks generally performed best with the ⟨3x3⟩ core which provides a higher frequency and manageable issue-width per thread. However in certain cases such as the computationally-intensive benchmark Ocean (and certain other cases such as LU and Stream), ConSSTEP actu-

ally benefits from slightly wider issue-widths for code phases which require higher ILP. Conversely, applications which are more memory intensive such as Barnes and Blackscholes slightly benefit from frequency gains since they are still limited by the memory system hierarchy.

9.2.5 Performance/Unit Area

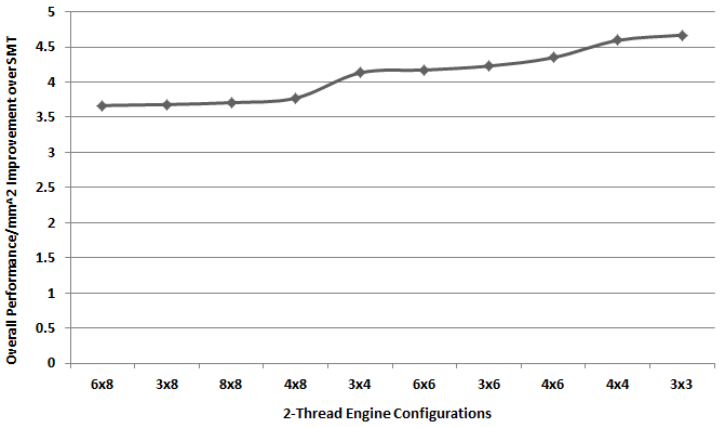


Figure 9.12: Two-Thread ConSSTEP Performance/mm^2 Improvement

Although this thesis work considers ConSSTEP as a single core architecture during experimental testing, assuming that performance for all CPUs and ConSSTEP scale fairly linearly as additional cores are integrated within a multi-core system, Fig. 9.12 displays ConSSTEP's instructions per second per square millimetre (performance/mm^2) improvement when compared to the SMT baseline processor. On average, 2T ConSSTEPs were able to improve the performance/mm^2 of SMTs by 4.09x. ⟨3x3⟩ in general provided the best performance/unit area, with the ⟨6x8⟩ and ⟨8x8⟩ providing the least improvement mainly due to frequency and area overhead in comparison to the smaller engines. Such engines however still provide a considerable performance/area gain when compared to a conventional SMT (greater than 3.5x).

9.2.6 Performance/Watt

Fig. 9.13 plots ConSSTEP's performance/watt improvement over a 2T SMT considering the same power constraint for both dies. According to the figure, ⟨3x3⟩ appears to be the most efficient mainly due to its speedup in throughput and low power consumption, followed close by the ⟨3x4⟩ core. ⟨3x3⟩ however demonstrates

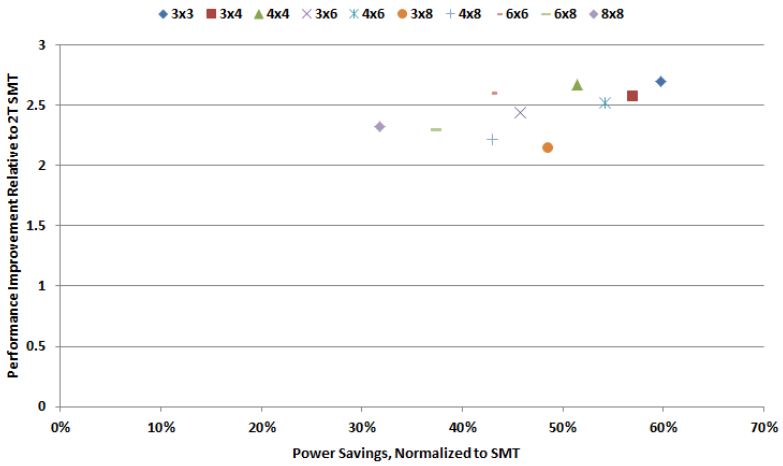


Figure 9.13: Two-Thread ConSSTEP Core Throughput/Power Consumption Improvement

an average 11.99% improvement over all other configurations, and therefore provides the most efficient performance/watt for a 2T core. In comparison to the SMT, all configurations demonstrate an average SMT performance/watt improvement of 1.18x.

Since ⟨8x8⟩ generally possesses the least gains for performance/area and performance/watt, the next section omits 8-FU engines and conducts further tests for 4-Thread configurations and efficient area, power, and performance per core.

9.3 Four-Thread Results

9.3.1 Area

Area comparisons for the 4T processors employ the same evaluation methodology as discussed for 2T, however considering a 4T pipeline, i.e. extra buffering, extended bit tags, and the thread select logic required for a 4T SMT versus 2T. Table 9.2 presents all processor die area results. The 4T SMT required less than a 1% area overhead for adding two more threads to the datapath. ConSSTEP's area overhead however increased linearly with the two threads added due to replicated engine logic and the scheduler which must monitor two more engines concurrently. Nevertheless, the ConSSTEP architecture is still able to reduce area overhead in the best case by 13.05% (i.e. ⟨3x3x3x3⟩), and in the worst case (i.e. ⟨6x6x6x6⟩) by 6.5% for a 4T core. The same transistor budget allocation for memory, scheduling etc. which

Table 9.2: 4-Thread Area Comparison (mm^2)

Processor	**Area (mm^2)**
4-Thread SMT	16.01
⟨3x3x3x3⟩	13.856
⟨3x4x3x3⟩	13.868
⟨3x3x4x4⟩	13.898
⟨3x3x3x6⟩	13.913
⟨3x3x4x6⟩	13.915
⟨4x4x4x4⟩	13.88
⟨3x4x4x6⟩	13.955
⟨3x3x6x6⟩	14.413
⟨4x4x4x6⟩	13.925
⟨3x4x6x6⟩	14.425
⟨4x4x6x6⟩	14.455
⟨3x6x6x6⟩	14.472
⟨3x4x4x4⟩	13.91
⟨6x6x6x6⟩	14.97

was discussed previously for 2T ConSSTEP scaled in the same manner for a 4T ConSSTEP core, except at a slightly larger die size. In addition to the overhead obtained by adding two additional thread's engines, Chapter 7's Table 7.3 demonstrates the additional unified scheduling logic required by ConSSTEP. Specifically, the scheduler's area nearly quadrupled to accommodate two extra threads, mainly attributed to the increased number of ports required by the scheduler to monitor all threads concurrently. Thus scheduler scalability remains future work.

9.3.2 Performance

IPC

Fig. 9.14 presents the *single-thread* IPC improvement of various 4T core models normalized to a 4T SMT, where the first column represents an average STC's IPC improvement within a 4T multi-core in comparison to a 4T SMT. Based on the results obtained, ConSSTEP provides an average *single-thread* IPC improvement of 1.418x over a 4-core STC, and 2.41x improvement in comparison to the 4T SMT.

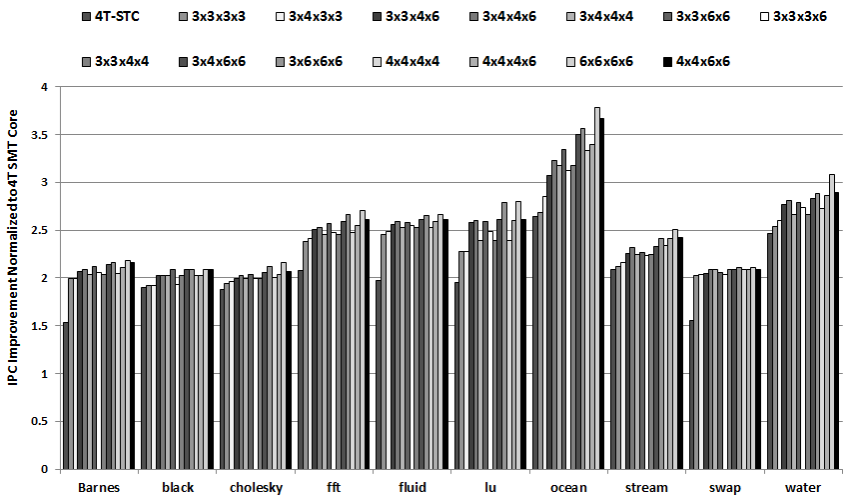


Figure 9.14: Four-Thread IPC for Various ConSSTEP Configurations

Similarly, the STC is able to provide a 2x the single-thread performance gain on average in comparison to its SMT equivalent. Thus ConSSTEP is able to provide a IPC performance gain in comparison to both the SMT core and STC.

Considering a 4T model, it is theoretically expected that the larger the engines present in a given ConSSTEP core, the greater the IPC gain as the core may support a greater issue width. In actuality, as previously discussed in the case of 2T cores, the PhysC's scheduling algorithms have an effect on heterogeneous configurations. Consequently, performance in certain cases do not follow the expected trends and demonstrate slightly less performance gains when compared to other homogeneous cores. Additionally, certain threads may also possess low inherent ILP and therefore when executed on larger engines do not exhibit substantial IPC gains in comparison to other configurations (i.e. the memory-intensive applications such as Black etc). For more computationally-intensive such as Ocean however, it is observed that cores with generally larger engines benefit from their higher inherent ILP per thread according to expected trends.

As seen in the figure, statistical observation concludes that the ⟨6x6x6x6⟩ on average provides the greatest gain due to larger engine size, issue width, and more resource availability for more computationally intensive applications. All ConSSTEP engines in general however were able to fully support the given workloads as threads scale, each engine exhibiting minimal pipeline contention in comparison to the SMT with the exception of certain memory operations. Conversely, for 4T workloads,

conventional SMT pipelines experienced greater contention delays as the number of threads in the pipeline scaled due to resource sharing and thread blocking. Consequently, a penalty in single-thread IPC performance was exhibited. For memory-intensive applications (SPr), and in even the worst case of Stream with high cache misses, the best ConSSTEP configuration was still able to achieve a 25.1% single-thread performance gain over 4T STC model due to the reduction of instruction processing overhead and misprediction penalties.

Throughput

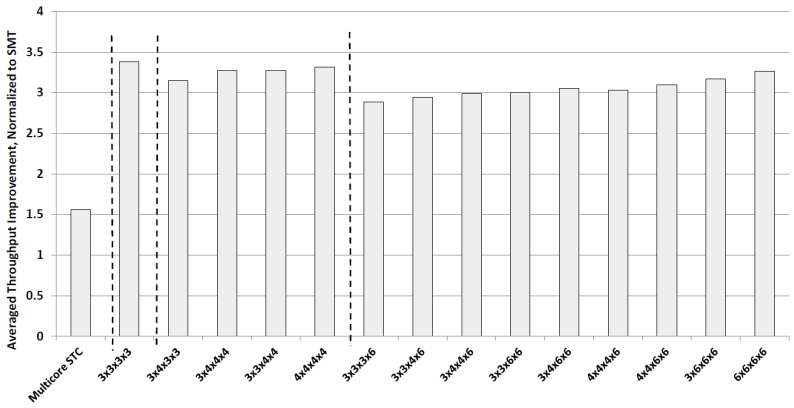


Figure 9.15: Four-Thread Throughput Comparison for Various ConSSTEP Configurations

Fig. 9.15 demonstrates the average throughput per 4T engine configuration, and the multicore STC throughput improvement over a 4T SMT equivalent. Similar to the 2T throughput section, the configurations are also divided in the figure according to the engine which limits attainable frequency of operation. The 4T trends exhibited in the figure follow the same trends as those of the 2T ConSSTEP cores, i.e. homogeneous cores achieve the best throughput results over heterogeneous cores as all engines operate at the maximum issue-width and frequency possible considering the given classification.

As seen in the results, the 4-core STC achieves 1.55x throughput improvement over a 4T SMT, where ConSSTEP engines on average achieve 3.13x the throughput of the SMT due to frequency, wider widths and reduced instruction processing overhead. In comparison to the 2T ConSSTEP configurations presented previously, a greater performance gain was also exhibited as all engines work concurrently with

minimal thread blocking in comparison to the SMT. ConSSTEP's overall limitation for additional performance gain however is directly related to the scheduler's scaling limitation of area overhead and power consumption. Therefore considering the maximum die size presented by the SMT, ConSSTEP is limited to 4-6 threads depending on the configuration, due to such scheduler scalability. Accordingly, this chapter now presents performance/unit area, where the scheduler issue is further discussed in Chapter 10.

9.3.3 Performance/Unit Area

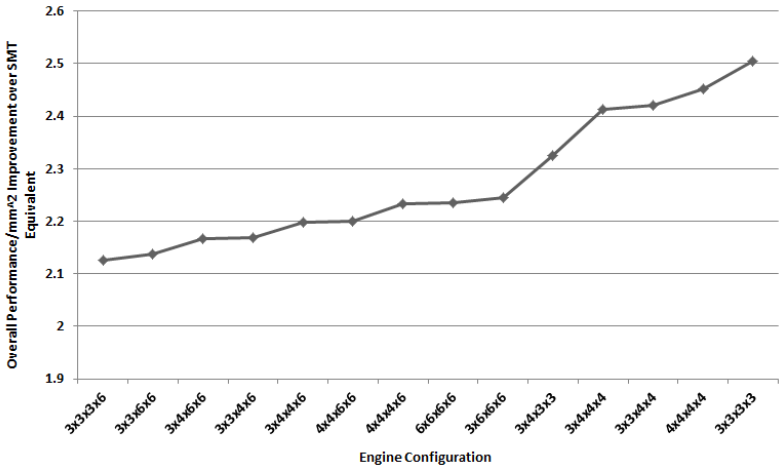


Figure 9.16: Four-Thread Performance/mm^2 Comparison for ConSSTEP Configurations

Fig. 9.16 displays instruction throughput per square millimetre (performance/mm^2), improvement when compared to the 4T SMT baseline processor equivalent. On average, 4T ConSSTEP was able to improve the performance/mm^2 by 2.27x, where ⟨3x3x3x3⟩ provided the best performance/unit area of 2.5x, and ⟨3x3x3x6⟩ and ⟨3x3x6x6⟩ provided the least improvement (however still greater than 2.2x). Such heterogeneous engine configurations were therefore limited by the largest engine, with direct correlation to throughput analysis discussed previously. The 4T configurations in general provided less of a performance/area improvement over the 2T configurations since the engines and scheduler doubled in area to scale the threads. Accordingly, ConSSTEP's performance/unit area gain was halved to support an additional two threads, nevertheless still achieving more than double the performance/unit improvement of an SMT, and magnitudes of improvement over a 4-core STC (i.e. four copies of the same core on a die versus a single ConSSTEP core).

9.3.4 Energy and Power

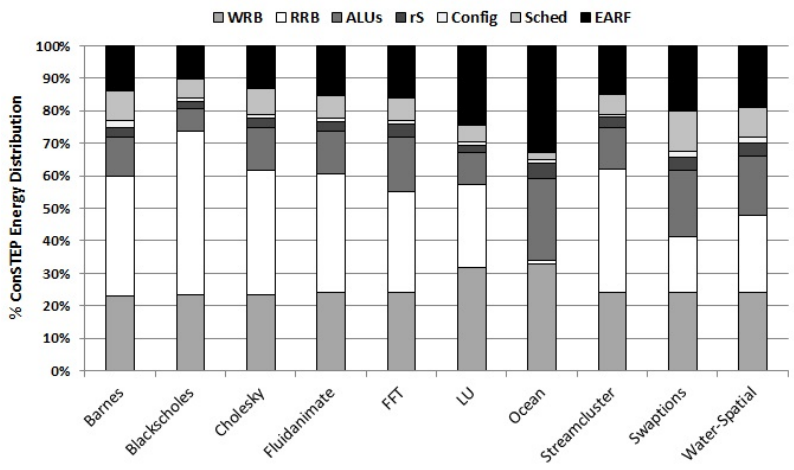


Figure 9.17: Four-Thread Energy Distribution for ConSSTEP Structures

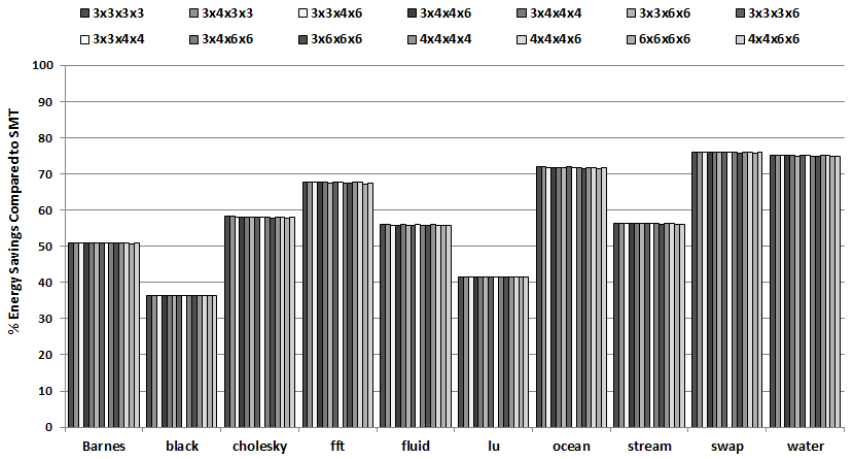


Figure 9.18: Four-Thread Energy Saving for Various ConSSTEP Engine Configurations versus SMT

Fig. 9.17 presents the energy distribution per ConSSTEP core structure per applications, averaged across all 4T ConSSTEP configurations. Similar to 2T cores, all configurations demonstrated the same relative energy consumption per component considering the overall system and equivalent operations required per benchmark. Accordingly consumption was also highly reliant on an application's characteristics for 4T cores. As expected, when directly comparing the 2T ConSSTEPs to 4T equivalents, the scheduler consumed a greater amount of energy. Since the scheduler monitored two more threads concurrently, more energy was required and hence a

greater portion of the total energy distribution was allotted to the scheduler. Aside from the scheduler's energy, all other structures remained similar to the 2T, consuming energy according to benchmark properties discussed previously.

Fig. 9.18 demonstrates the energy savings of a 4T ConSSTEP core in comparison to a 4T SMT, displaying similar, slightly varying energy consumption trends per core type (as the case of the 2T ConSSTEP). 4T ConSSTEP cores save on average approximately 58.93% energy in comparison to the 4T SMT. Specifically, the 4T cores increase energy/application efficiency by approximately 1.57x in comparison to an SMT core equivalent. Such improvement is again attributed to the speedup in execution time and the non-blocking characteristics of the engines as threads scale, whereas contention and blocking increases in SMT processors as threads scale, increases the execution time and energy required per thread. However since the number of structures increases within a ConSSTEP core as threads scale, power consumption likely also increases as discussed next.

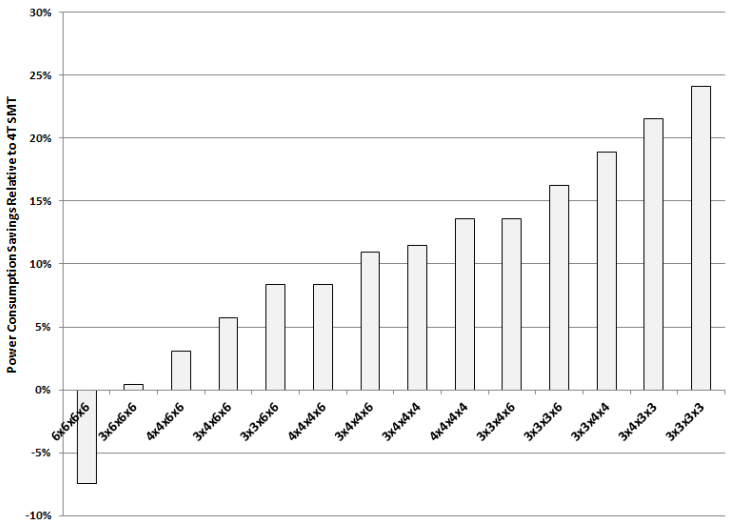


Figure 9.19: Power Consumption Savings for 4T ConSSTEP Configurations, Normalized to SMT

Fig. 9.19 presents the dynamic and static power savings for various 4T ConSSTEP configuration, normalized to its SMT core equivalent. In the case of 4T cores, results demonstrate that the most efficient configuration is the ⟨3x3x3x3⟩ and ⟨3x4x3x3⟩ core, saving approximately 24% and 22% power when compared to the SMT, respectively. In the case of 4T cores however, more power is consumed in

comparison to the 2T versions due to the additional logic required to scale ConSSTEP to four threads. Thus power savings are less than the savings discussed for 2T, nevertheless still saving a conservative amount of power when compared to a standard SMT. Conversely, the ⟨6x6x6x6⟩ configuration consumes more power than the SMT and hence this irregularity is observed in the figure; the irregularity is due to the increased number of integer-based FUs and the scheduler. Similarly the ⟨3x6x6x6⟩ and ⟨4x4x6x6⟩ consume nearly the same power as the SMT, for similar reasons, i.e. more 6-FU engines present on the core and the monitoring required by the scheduler. An average power savings of 11% however exists across all 4T configurations.

9.3.5 Performance/Watt Comparison

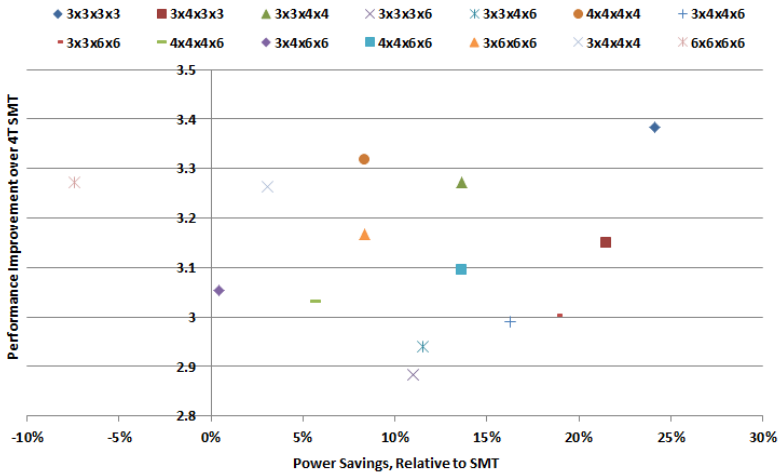


Figure 9.20: Four-Thread Throughput/Power Comparison for ConSSTEP Configurations

Fig. 9.20 plots the throughput improvement/watt savings normalized to the 4T SMT, displaying the most efficient configurations in terms of performance and power, for various 4T ConSSTEP configurations. According to the plot, ⟨3x3x3x3⟩ provides the best performance per Watt due to its increased frequency and smaller engine sizes, whereas ⟨6x6x6x6⟩ yield the least improvement as expected based on the power outcomes presented in the previous subsection. Overall, the ⟨3x3x3x3⟩ is on average 2x more efficient than the other configurations presented.

9.4 Other Architectural Statistics

9.4.1 Configuration Memory

Table 9.3: Configuration Memory Specifications

Config Mem	**Timing Specifications**	**Total Storage(KB)**
rS	5 c.c/rS	240
FUs	2 c.c/FU	120
RRB	4 c.c	20
WRB	4 c.c	4
(Pre-) Load Memory	2-3 c.c/load	2
Extern Reg File	2 c.c/two reads (multi-port)	1

Many conventional pipeline structures are eliminated in ConSSTEP through the use of configuration data and simple engine structures i.e. the IQ, ROB, Rename, ID, Bypass etc. Therefore this section assesses the requirements imposed by such configuration data storage. The following outlines the average configuration (banked) memory storage requirements per engine component:

- FUs: Configured in twos - 16Kb (i.e. 8Kb per ALU)
- rS: Configured in twos - 16Kb
- RRB: 8Kb
- WRB: 1Kb
- EARF R/W: 0.25 Kb
- Pre-Load Memory: 0.5kB

The *total* memory requirements per engine component (aggregated storage) are also outlined in Table 9.3 and are averaged per application. Assuming larger and more intensive applications than the benchmarks presented here, the configuration memories must then be treated as data/instruction caches, transferring configuration data with multiple levels of cache. Overall, the statistics presented here demonstrate that the scalability and reconfigurability of the engines and interconnect come with the cost of approximately 5KB per additional row added to an engine. Therefore, ConSSTEP requires more C-Cache data storage in comparison to a conventional CPU's I-Cache, however at the advantage of eliminating various *dynamic* pipeline

structures while providing performance gains and an energy efficient design.

9.4.2 Revisiting Tseng and Patt with ConSSTEP

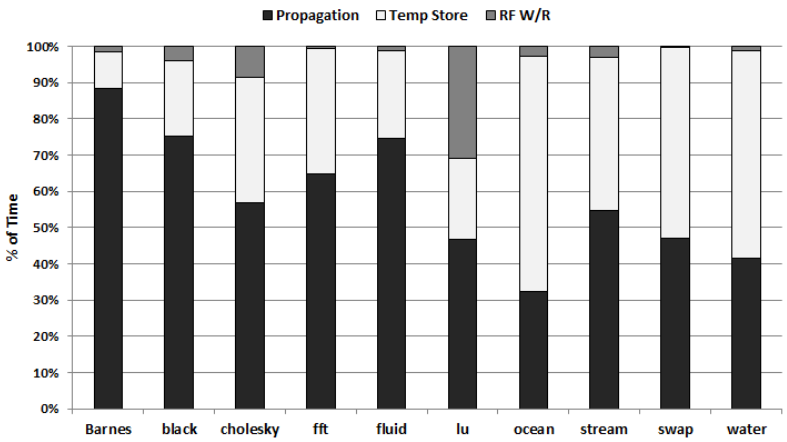


Figure 9.21: Propagation, Temporary Storage, and External Read/Write Requirements of Operand Dependencies with ConSSTEP

One of the prime objectives of ConSSTEP is to eliminate data transport issues by exploiting value lifetime. This section revisits and analyzes ConSSTEP's storage and propagation statistics, and compares them to the findings presented by Tseng and Patt [6]. Fig. 9.21 demonstrates the overall operand dependency characteristics observed by ConSSTEP, averaged across all 2T and 4T configurations for each benchmark. Specifically, the figure displays the percentage of instruction operands which require propagation, temporary storage on the rS network, and external register file storage during bundle execution. Referring back to the benchmark statistics gathered by Tseng and Patt [6], 70% of the broadcasts in conventional processors which are sent to the IQ are not required as they have been forwarded to their single consumer using the bypass network. Similarly, 74% of the results stored to the register file have already been forwarded to their single consumer, never read, and/or overwritten by other instructions, where 80% of the values have a lifetime of 32 instructions or fewer.

According to the statistics obtained by ConSSTEP during experimental testing, approximately 58.25% of the operands were directly propagated to their consumers throughout the network, 36.33% of the operands required temporary storage for an average of two instructions or less, and on average 5.42% of the operands required

external register file storage due to inter-bundle dependencies. Therefore by using the ConSSTEP approach, operand lifetimes were able to be maintained within the bundles, allowing approximately 94.58% of instruction operands to be directly forwarded to their consumer(s) without register file intervention (whether directly propagated or briefly stored in the engine), where only 5.42% of the operands required EARF reads and writes. In comparison to the statistics obtained by Tseng and Patt, ConSSTEP therefore improves operand forwarding by 20.58% due to register file access avoidance during bundle execution, effectively minimizing IO register port requirements, its centralized access, and the general unscalability of the bypass network.

9.4.3 Load/Store Unit Scaling

Given that the number of FUs per engine have been increased in ConSSTEP to obtain a computational performance gain in comparison to conventional processors, it was observed that various benchmarks are further bound by memory operations. To view the effects of increasing concurrent memory operations in ConSSTEP, Fig. 9.22 presents the IPC, cycle time, and energy/access increase by scaling the number of LSUs per engine which access the cache, averaged for all benchmarks. The figure displays trends for a 2T ConSSTEP core, normalized to 2 LSUs/engine for cycle time and energy access, where IPC is normalized to a STC. Cycle time (ns) and energy per access (nJ) were both obtained using CACTI estimates.

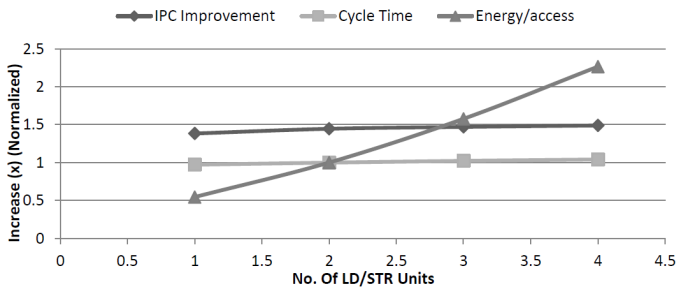


Figure 9.22: Two-Thread ConSSTEP LSU Scaling

As seen in Fig. 9.22, an average performance improvement of 4% was obtained by using 2 LSU/engine in comparison to one LSU. Adding an additional LSU/engine only displayed 1% further IPC improvement due to contention, where two more LSUs only provided a 0.75% performance increase. On average, cycle time increased at a

slow linear rate, whereas energy/access increased rather exponentially. Accordingly, the inclusion of two LSUs per engine (multiplexed to a 2-write port D-cache) provided fair performance gains while maintaining energy/access of the cache within reasonable bounds.

9.5 Summary

9.5.1 2-Thread ConSSTEP

According to the results obtained, the most efficient 2T ConSSTEP engine configuration was the ⟨3x3⟩ core due to its small relative size, low power consumption, higher frequency, and low area overhead. Its minimized critical path was therefore able to increase throughput in comparison to the other configurations tested, however at the expense of a lower issue width in comparison to a 2T STC. ⟨3x3⟩ however demands less of other architectural structures such as configuration memory, FUs, and general PhysC complexity. In the case that the frequency of operation may be further optimized however (whether by variable frequency per engine or deeper pipelining techniques within each engine), and that area overhead may be further mitigated, the larger configurations such as ⟨4x6⟩, ⟨4x8⟩ also appear as promising solutions for increasing issue width and the general performance of a core. However, considering the architectural design presented in this thesis, the most efficient 2T ConSSTEP configuration was determined to be the ⟨3x3⟩ core.

9.5.2 4-Thread ConSSTEP

Similar to the 2T case presented, based on the results obtained the most efficient 4T ConSSTEP configuration evaluated was the ⟨3x3x3x3⟩ engine architecture. In accordance with the reasons mentioned above for 2T, as threads scale on a ConSSTEP core, area overhead and power consumption also increased. Consequently the larger the engines and the more threads supported per core, the greater the area and power overhead required. However as seen in the results, ConSSTEP nevertheless presented a more performance scalable solution than the SMT as thread blocking is still problematic in such a model.

9.6 Conclusion

This chapter provided experimental results for the ConSSTEP architecture, with detailed comparisons to the SMT and STC multi-core baselines. Specifically this chapter raised various questions pertaining to performance, energy efficiency, power consumption, and area overhead of ConSSTEP to determine the results and feasibility of the core. The following is a summary of the findings based on the 2T and 4T core prototypes tested:

- **Performance:**
  - **IPC:** On average, ConSSTEP was able to improve single-thread IPC of a 2T SMT by 1.39x, and achieve a 1.11x single-thread performance gain over STC models. Scaling the core to 4T, ConSSTEP improved IPC by 2.41x in comparison to a 4T SMT and 1.41x the single-thread IPC STC. ConSSTEP was therefore successful at mitigating much of the thread blocking imposed of typical SMTs, while eliminating much of the processing overhead per instruction, which is especially effective during speculative execution and respective single-thread performance.
  - **Throughput:** Considering the increase in attainable frequency of operation, an average 2T ConSSTEP core was able to achieve 2.4x the throughput of a SMT, whereas a 4T core achieved 3.13x the throughput as threads scaled. Hence the simple logic which increased ConSSTEP's frequency greatly contributed to ConSSTEP's performance increase, especially for non memory-intensive applications. Memory-intensive applications however were also able to receive slight benefits from the increase in frequency and/or issue-width attained by ConSSTEP.
- **Energy Efficiency:** Through experimental results, ConSSTEP was able to achieve 37.5% more energy efficiency when compared to a 2T SMT, and 58.93% when compared to a 4T SMT. In comparison to a STC multi-core system which duplicates cores on a die, ConSSTEP would therefore exhibit exponential energy efficiency.
- **Power consumption:** Considering all active and inactive components on chip, ConSSTEP achieved 59% power savings in comparison to 2T SMT, and on average 11% for 4T SMT, mostly due to component duplication and scheduler complexity of the ConSSTEP core.

- **Area overhead:** According to the physical model results obtained for the cores, ConSSTEP was able to reduce area overhead of a 2T SMT die on average between 52.6% - 56.3%, and 6.5 - 13.05% for a 4T SMT. Due to component duplication and the scalability issues of the scheduler, ConSSTEP was limited to 4 threads per core. ConSSTEP however greatly increased performance while providing non-blocking thread behaviour in multi-threaded workloads when compared to conventional processor cores for the given die area.

- **Performance/Unit Area:** According to the results obtained from the simulator and physical modelling, considering the same area per core, an average 2T ConSSTEP core was able to achieve 4.09x the performance/unit area of a 2T SMT, and 2.27x that of a 4T SMT.

- **Performance/Watt:** Similarly, considering the same power constraint per die, ConSSTEP on average achieved 1.18x the performance/watt of a 2T SMT, and 2x that of a 4T SMT.

Therefore according to the experimental results presented in this chapter, ConSSTEP is a feasible solution to increase a SMT's single-thread performance, throughput, and energy efficiency with generally less area overhead per die. ConSSTEP is able to support up to 4 threads considering a conventional SMT's maximum die size.

Chapter 10

Conclusions and Future Work

Increasing both single-thread performance and throughput on a single-core is a challenging and difficult task, especially considering issues of hardware complexity and power consumption. This dissertation introduced the ConSSTEP core which completely deviated from conventional processor designs, integrating reconfigurable properties to allow for higher issue widths, the elimination of several datapath bottlenecks, scalable data transport, and higher single-thread performance and energy efficiency on a single chip.

ConSSTEP is a configurable multi-threaded processor which employs the concept of logical and physical compilation to support an underlying reconfigurable architecture. Specifically, the core consists of a variety of execution units (engines), each containing functional units (FUs) interconnected through a configurable, single-cycle multi-hop registerSwitch (rS) interconnect. The rS units solve the data transport bottleneck of conventional processors by providing both temporary distributed storage and data propagation properties, relieving the need for constant access to a register file, bypass network, and instruction queue.

Each thread of a multi-threaded workload receives a dedicated engine, where multiple engines work concurrently on a chip to provide high instruction throughput. Thus single-thread performance remains unaffected by resource contention as the case of other multi-threading models. The ConSSTEP architecture and physical compilation technique therefore were successful at increasing both single-thread performance and the throughput of multi-threaded workloads.

Accordingly, the following answer the *Research Questions* raised in Chapter 1.2:

Research Question 1: What nuanced computing model and datapath design may mitigate instruction control and processing overhead to emphasize data processing and increase a single-thread's performance? How can the model be applied to a multi-threaded workload domain to improve upon a SMT model's limitations?

Answer 1: A configurable datapath model, referred to as ConSSTEP, was defined as a nuanced computing model. The processor successfully mitigated instruction control and processing overhead, where the datapath was completely altered to eliminate several conventional core and SMT model limitations. The ConSSTEP model effectively increased single-thread performance, and was directly applied to the multi-threaded workload domain which significantly improved area, performance, and energy efficiency per core.

Research Question 2: Is it possible to alter the triple set characteristics of a Computing System with Programmable Procedure (CSPP) in order to mitigate the effects of redundant bandwidth, unscalable structures, and the general bottlenecks raised in Research Question 1? If the triple set is altered, how will this affect the ISA and software/compiler compatibility?

Answer 2: The triple set characteristics of the Computing System with Programmable Procedure (CSPP) model was re-defined successfully as $A_{ConSSTEP} = \{C_i, \sim L_{i,j}, \sim P_{i,j}\}$. Configurable properties were integrated into the datapath, i.e. flexible links ($\sim$L) interconnecting the functional components (C), which mitigated several conventional CPU effects including redundant bandwidth, buffering, instruction processing overhead, and unscalable hardware structures. Configurable bottleneck properties (i.e. additional latencies, hardware overhead, configuration memory) were decreased by taking these factors into consideration during the datapath's design. Specifically, configuration latencies were eliminated using a distributed configuration banking system and an aggressive pipeline with simultaneous execution and configuration. Hardware overhead was also eliminated by integrating configurable logic within the processor's design. However additional configuration memory requirements were necessary in place of the Instruction cache (I-cache).

In terms of software/compiler compatibility, the reconfigurable triple set required a secondary compilation phase, referred to as the Physical Compiler (PhysC) to generate the configuration logic. The first phase referred to as a Logical Compiler, i.e.

any standard compiler, was kept to maintain software and binary compatibility. This the compilation phase provided an application binary, where the binary was then input to the PhysC generating bundles of instructions which were translated to configuration logic. Such logic was then used by the underlying architecture for execution of bundles based on their respective dependencies. Therefore current ISAs and software/compilers remain fully compatible using ConSSTEP.

Research Question 3: Is it possible to have a smarter compilation process to extract application characteristics and eliminate various datapath structures? If so, is it possible to maintain compatibility with standard software, ISAs, and programming models? If compatibility is maintained, how will the software interact with the processor's hardware to convey such application characteristics?

Answer 3: As mentioned in Answer 2, a two layer compilation process was developed in this thesis to successfully extract application characteristics within the compiler, eliminating several front-end datapath structures. Accordingly such a framework required no ISA amendments/extensions and maintained full software compatibility. The software was able to interact and convey application characteristics to the underlying processor hardware through reconfigurable logic and its respective hardware properties. Specifically, as ConSSTEP invokes a configurable interconnect, the PhysC generates the core's configuration data based on the given workload's communication characteristics and data dependencies. This data conveys the application's requirements to the underlying hardware as simple control logic, eliminating the need for conventional structures to re-discover application characteristics at the cost of area, power and additional process latencies.

Research Question 4: Once the processor is built, how will exceptions and/or mispredictions be handled?

Answer 4: Based on ConSSTEP's architecture, mispredictions and exceptions were designed to be handled in a simplistic and efficient manner, especially when directly compared to conventional processors. Exceptions are handled through the technique of "pausing" which uses the double configuration register approach as introduced in this thesis. Specifically, when an exception is raised by the executing configuration register set, the alternate set may be reconfigured to execute the exception handler while the executing set is simply *paused* and resumed after the ex-

ception is resolved. Similarly, mispredictions use the alternate set to verify that the misprediction is not a loop. If the misprediction is not a loop, then a reconfiguration penalty is endured. Else the misprediction penalty is completely avoided. However both approaches completely eliminate the need to flush the pipeline and checkpoint various structures i.e. store and restore state prior to the exception/misprediction. Thus ConSSTEP handles exceptions simplistically and efficiently.

10.1 Addressing Limitations

Although there were many advantages to the ConSSTEP architecture, limitations did exist as discussed in Chapter 3 and during experimental results. Therefore this section presents and lists possible solutions for addressing such limitations in future work:

Scheduler Scalability: This thesis presented a unified scheduler approach for monitoring thread workloads and distributing bundles to each thread's engine concurrently. A unified scheduler was especially required to provide thread synchronization across the threads and to simplify exception handling. As seen during experimental results however, such a unified approach led to double the area and power consumption for every two threads added to the system, limiting massive scalability on chip. Future work therefore involves a design which provides scheduler scalability in ConSSTEP.

A possible solution to allow for such scheduler scalability may involve adding dedicated synchronization mechanisms to monitor barriers, while integrating individual thread schedulers per engine. Similarly, a dedicated exception handling unit could be included in the core, however in a scalable manner without IO port explosion. Although such a scheduler design could possibly benefit ConSSTEP, a core's scalability would thereafter depend on the memory system bottleneck, as the case of conventional CPUs.

Sharing FP Units Between Engines: A minor scalability issue in ConSSTEP involved FP engines. Specifically, a FP unit was assigned to each engine/thread to ease the PhysC's scheduling complexity. As seen in the experimental results provided however, such FP logic added to the area overhead and power consumption per engine. FP operations in general were also infrequent across the applications.

Therefore a more ideal solution would be to share FP engines between integer engines for greater area and power scalability. A dynamic protocol between the integer WRB to the FP RRB would therefore be required to implement such sharing tactics, and remains future work.

Runahead Techniques for Improving Memory Performance: The memory system invoked by ConSSTEP was implemented in an identical manner to the baseline cores to directly assess the benefits of a nuanced processor architecture. Experimental testing demonstrated that ConSSTEP was successful at increasing the performance of multi-threaded workloads, with a slight performance advantage for more memory-intensive/SPr applications. Given that ConSSTEP mitigates the many impositions of pipeline flushes and state restoring, it would be possible to increase its memory performance using runahead thread techniques[17, 18].

Runahead threads allow a conventional pipeline to continue executing instructions with a fake "load/store" value when the oldest instruction in the pipeline is a L2 miss (or locks in the case of multi-threaded workloads). The instruction stream continues to execute with the fake value, where future load/store instructions are issued in order to prefetch data and mitigate future misses. Context must be saved in runahead mode however so that the previous state (prior to the fake value) may be restored once the L2 miss returns. Thus all instructions which execute after the L2 miss must be squashed, which consumes dynamic power for the possibility of data prefetching, where the pipeline must be restored regardless of the value used for runahead.

Similar to the case of exception handling, storing, restoring, and flushing pipeline latencies are costly to process performance. Since ConSSTEP possesses much simpler "pausing" logic to handle such situations faster, using a revised runahead method would prove promising without the need to save/restore state for prefetching techniques and further performance gains in memory-intensive workloads. Hence a runahead ConSSTEP implementation remains future work.

10.2 Future Work

The following presents other possible solutions to further improve the quality and functionality of the ConSSTEP processor:

SIMD Support: This thesis directly addresses issues of ILP and TLP, leaving Data-Level Parallelsim (DLP) as future work, i.e. the integration of SIMD units in the core. It is possible that a technique such as that employed for the FP engine may be used to integrate SIMD units, however in a shared manner. Direct consideration of the memory banking requirements for SIMD and general vector processing must also be investigated. It is possible that the engine units themselves may also be vectorized, invoking rS unit techniques for temporary storage. However due to such complexities and design issues, SIMD support remains future work.

Apply core morphing to ConSSTEP: Similar to the CoreFusion [57] and MorphCore [10] approaches discussed, ConSSTEP may apply dynamic thread monitoring in the core to temporarily shut down engines, or fuse engines together to provide higher issue-width for more computationally-intensive workloads such as Ocean.

Revising the primary compiler to avoid unnecessary register spilling: As ConSSTEP was successful at mitigating register file accesses and exploiting operand lifetimes using engines and the rS interconnect, it is possible that loads and stores due to register spilling may be tolerated in a more efficient manner by the primary compiler. That is, reducing the memory accesses generated by the compiler due to the insufficient architectural registers present in the ISA. Although a fairly complex endeavour, such a compilation approach remains future work.

Multi-core adaptability: The ConSSTEP architecture presented in this work, considers a single-core multi-threaded processor. However ConSSTEP may easily be adapted to a multi-core design to provide more significant performance gains. Since the memory system invoked by ConSSTEP core is identical to that of conventional cores, the microarchitectural design presented in this thesis may easily support the same coherence protocols without any architectural modifications. Accordingly the ConSSTEP cores presented in this thesis work may be applied to heterogeneous and/or homogeneous multi-core systems without amendments and remains future work.

Bibliography

[1] S. Borkar and A. Chien. The future of microprocessors. In *Communications of the ACM*, volume 54, pages 67–77, 2011.

[2] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. In *IEEE Computer*, pages 33–38, 2008.

[3] P. Michaud, A. Mondelli, and A. Seznec. Revisiting clustered microarchitecture for future superscalar cores: A case for wide issue clusters. In *ACM TACO*, volume 13, pages 22–33, 2015.

[4] Y. Patt. Future microprocessors: What must we do differently if we are to effectively utilize multi-core and many-core chips. In *IPSI BGD Transactions on Internet Research*, volume 5, pages 2–10, 2009.

[5] L. Kirischian. *Reconfigurable Computing Systems Engineering: Virtualization of Computing Architecture*. Taylor & Francis, 2016.

[6] F. Tseng. Braids: Out-of-order performance with almost in-order complexity. In *Doctoral Dissertation, University of Texas at Austin*, 2008.

[7] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[8] H. Corporaal. Design of transport triggered architectures. In *Design Automation of High Performance VLSI Systems*, pages 130–135, 1994.

[9] E. Athanasaki, N. Anastopoulos, K. Kourtis, and N. Koziris. Exploring the performance limits of simultaneous multithreading for memory intensive applications. *Journal of Supercomputing*, 44:44–64, 2008.

[10] Khubaib, M. Aater Suleman, M. Hashemi, C. Wilkerson, and Y. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Proceedings of MICRO*, pages 305–316, 2012.

[11] S. Everman and L. Eeckhout. A memory-level parallelism aware fetch policy for smt processors. In *Proceedings of HPCA*, pages 240–249, 2007.

[12] C. Bienia, S. Kumar, J. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of ACM PACT*, pages 72–81, 2008.

[13] N. Barrow-Williams, C. Fensch, and S. Moore. A communication characterisation of splash-2 and parsec. In *Proceedings of IISWC*, pages 86–97, 2009.

[14] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in smt processors. In *IEEE High Performance Computing Architectures (HPCA)*, 2003.

[15] D. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of MICRO*, pages 318–327, 2001.

[16] K. Van Craeynest, K. Eyerman, and S. Eeckhout. Mlp-aware runahead threads in a simultaneous multithreading processor. In *Proceedings of HiPEAC*, pages 110–124, 2009.

[17] T. Ramirez, A. Pajuelo, O. Santana, and M. Valero. Runahead threads to improve smt performance. In *Proceedings of HPCA*, pages 16–20, 2008.

[18] T. Ramirez, A. Pajuelo, O. Santana, O. Mutlu, and M. Valero. Efficient runahead threads. In *Proceedings of PACT*, pages 443–452, 2010.

[19] S. Eyerman and L. Eeckhout. The benefit of smt in the multi-core era: Flexibility towards degrees of thread-level parallelism. In *Proceedings of ASPLOS*, pages 591–606, 2014.

[20] J. Baer. Microprocessor architecture: From simple pipelines to chip multiprocessors. In *Cambridge University*, 2010.

[21] J. Hennessy and D. Patterson. Computer architecture: A quantitative approach. Morgan Kaufmann Publishers Inc: San Francisco, 2011.

[22] T. Jamil. Risc versus cisc. In *IEEE Potentials*, volume 15, pages 13–16, 1995.

[23] K. Karuri and R. Leupers. *The ASIP Design Space*. Springer New York, 2011.

[24] K. Beyls and E. H. D'Hollander. Generating cache hints for improved program efficiency. In *Journal of Systems Architecture*, volume 51, pages 233–250, 2005.

[25] J. Shen and M. Lipasti. Modern processor design: Fundamentals of superscalar processors. In *McGraw-Hill Publishers*, 2004.

[26] P. Y Chang, M. Evers, and Y. N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *Porceedings of Parallel Architectures and Compilation Techniques (PACT)*, pages 48–58, 1996.

[27] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. In *Journal of Instruction Level Parallelism (JILP)*, pages 1–23, 2006.

[28] J. Hennessy. Processor design and other challenges in the post-pc era. In *Proceeding of Microprocessor Forum,, Cahners Microdesign Resources*, 1999.

[29] Intel 64 and ia-32 architectures optimization reference manual. In *Technical Report 248966-026*, 2012.

[30] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *International Symposium on Microarchitecture*, pages 291–302, 2003.

[31] A. Pellegrini, J. Greathouse, and V. Bertacco. Viper: Virtual pipelines for enhanced reliability. In *Proceedings of the International Symposium on Computer Architecture*, pages 344–355, 2012.

[32] E. B. Nightingale, J. R Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer pcs. In *Proceedings of EuroSys*, pages 343–356, 2011.

[33] R. Nagarajan. Design and analysis of technology scalable architectures. In *PhD Thesis - University of Texas at Austin*, 2006.

[34] K. Sankaralingam. Polymorphous architectures: A unified approach for extracting concurrency of different granularities. In *PhD Thesis - University of Texas at Austin*, 2006.

[35] S. Kalathingal, S. Collange, B. N. Swamy, and A. Seznec. Dynamic inter-thread vectorization architecture: Extracting dlp from tlp. In *Proceedings of SBAC-PAD*, 2016.

[36] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *ACM SIGARCH Computer Architecture News*, pages 82–91, 1990.

[37] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Dubiatowicz, B. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: Architecture and performance. In *International Symposium on Computer Architecture (ISCA)*, pages 1–12, 1995.

[38] P. Kontegira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. In *IEEE MICRO*, pages 21–29, 2005.

[39] R. Kalla, B. Sinharoy, and J. M Tendler. Ibm power5 chip: A dual-core multithreaded processor. In *IEEE Micro: Hot Chips*, volume 24, pages 40–48, 2004.

[40] J. Chiu, Y. Chou, and P. Chen. Hyperscalar: A novel dynamically reconfigurable multi-core architecture. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 277–285, 2010.

[41] D. Tarjan M. Boyer and K. Skadron. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the Design Automation Conference (DAC)*, pages 772–775, 2008.

[42] M. Qayum, N. Siddique, M. Haque, and A. S Tayeen. Future of multiprocessors: Heterogeneous chip multiprocessors. In *International Conference on Informatics, Electronics and Vision (ICIEV)*, pages 372–376, 2012.

[43] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 449–460, 2012.

[44] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of CGO*, pages 1–10, 2003.

[45] V. Packirisamy, Y. Luo, W. Hung, A. Zhai, P. C. Yew, and T. Ngai. Efficiency of thread-level speculation in smp and cmp architectures - performance, power, and thermal perspective.

[46] M. Ros and P. Sutton. A hamming distance based vliw/epic code compression technique. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 132–139, 2004.

[47] H. Sharangpani and H. Arora. Itanium processor microarchitecture. In *IEEE MICRO*, volume 20, pages 24–43, 2000.

[48] *NVIDIA Charts Its Own Path to ARMv8*, 2014.

[49] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[50] E. Gunadi and M. Lipasti. Crib: Consolidated rename, issue, and bypass. In *International Symposium on Computer Architecture (ISCA)*, pages 23–32, 2011.

[51] R. P Goldberg. Survey of virtual machine research. In *IEEE Computer*, volume 7, pages 34–45, 1974.

[52] S. Hilly and A. Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proceedings of HPCA*, pages 64–67, 1999.

[53] K. Sankaralingam, R. Nagarajan, H. Lui, C. Kim, J. Huh, N. Ranganathan, D. Burger, and S. Keckler et al. Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp. In *ACM TACO*, 2004.

[54] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, and et al. Task superscalar: An out-of-order task pipeline. In *IEEE/ACM International Symp. on Microarchitecture (MICRO)*, pages 89–100, 2010.

[55] Y. Watanabe, J. D. Davis, and D. A. Wood. Widget: Wisconsin decoupled grid execution tiles. In *International Symposium on Computer Architecture (ISCA)*, pages 2–13, 2010.

[56] V. Govindaraju, C. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *IEEE MICRO*, volume 32, pages 503–514, 2011.

[57] E. Ipek, M. Kiman, N. Kiman, and J. F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of ISCA*, pages 186–197, 2007.

[58] Barcelona Supercomputing Center. Programming with ompss. In *BSC Programming Models - Technical Document*, 2017.

[59] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou. A fully pipelined and dynamically composable architecture of cgra. In *IEEE Symp on FPGA Custom Computing Machines (FCCM)*, pages 9–16, 2014.

[60] E. Mirsky and A. Dehon. Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *IEEE Symp on FPGA Custom Computing Machines (FCCM)*, pages 137–166, 1996.

[61] S. Goldstein, H. Schmit, M. Budio, and et al. Piperench: A reconfigurable architecture and compiler. In *IEEE Computer*, volume 33, pages 70–77, 2000.

[62] Z. A. Ye, A. Moshovos, S. Hauch, and P. Banerjee. Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *International Symposium on Computer Architecture (ISCA)*, pages 225–235, 2000.

[63] M. A. Watkins and D. H. Albonesi. Remap: A reconfigurable architecture for chip multiprocessors. In *IEEE MICRO*, pages 65–77, 2011.

[64] R. K. Pal, K. Paul, and S. Prasad. Rekonf: A reconfigurable adaptive manycore architecture. In *IEEE Parallel and Distributed Processing with Applications (ISPA)*, 2012.

[65] J. H. Ahn, M. Erez, and W. J. Dally. Tradeoff between data-, instruction-, and thread-level parallelism in stream processors. In *Proceedings of ICS*, pages 126–137, 2007.

[66] W. J. Dally, P. Hanrahan, M. Erez, and T. J. Knight. Merrimac: Supercomputing with streams. In *Proceedings of ACM/IEEE Conference on SC*, page 35, 2003.

[67] H. Corporaal and M. Arnold. Using transport triggered architectures for embedded processor design. In *Integrated Computer-Aided Engineering*, pages 19–38, 1998.

[68] S. Shahabuddin, J. Janhunen, and M. J. Juntti. Design of a transport triggered architecture processor for flexible iterative turbo decoder. *CoRR*, abs/1501.04192, 2014.

[69] G. De Micheli and L. Benini. *Networks on Chips: Technology and Tools*. Morgan Kaufmann Publishers Inc., 2006.

[70] G. McFarland. Microprocessor design: A practical guide from design planning to manufacture. In *McGraw-Hill*, 2006.

[71] N. Binkert, B. Beckmann, and G. Black et al. The gem5 simulator. In *SIGARCH Computer Architecture News*, volume 39, pages 1–7, 2011.

[72] N. Muralimanohar, R. Balasubramonian, and N. P Jouppi. Cacti 6.0: A tool to understand large caches. In *Technical Report, Hewlett-Packard Labs*, 2008.

[73] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. In *Proceedings of the IEEE*, volume 89, pages 490–504, 2001.

[74] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson - Addison Wesley, 2005.

[75] S. Shankland. Power could cost more than servers, google warns. In *CNET web article*, 2006.