HIGH PERFORMANCE ARCHITECTURE USING SPECULATIVE THREADS AND DYNAMIC MEMORY MANAGEMENT HARDWARE

Wentong Li

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

December 2007

APPROVED:

Krishna Kavi, Major Professor and Chair of the Department of Computer Science and Engineering
Phil Sweany, Minor Professor
Robert Brazile, Committee Member
Saraju P. Mohanty, Committee Member
Armin R. Mikler, Departmental Graduate Coordinator
Oscar Garcia, Dean of College of Engineering
Sandra L. Terrell, Dean of the Robert B. Toulouse School of Graduate Studies

Li, Wentong, High Performance Architecture using Speculative Threads and Dynamic Memory Management Hardware. Doctor of Philosophy (Computer Science), December 2007, 114 pp., 28 tables, 27 illustrations, bibliography, 82 titles.

With the advances in very large scale integration (VLSI) technology, hundreds of billions of transistors can be packed into a single chip. With the increased hardware budget, how to take advantage of available hardware resources becomes an important research area. Some researchers have shifted from control flow Von-Neumann architecture back to dataflow architecture again in order to explore scalable architectures leading to multi-core systems with several hundreds of processing elements.

In this dissertation, I address how the performance of modern processing systems can be improved, while attempting to reduce hardware complexity and energy consumptions. My research described here tackles both central processing unit (CPU) performance and memory subsystem performance. More specifically I will describe my research related to the design of an innovative decoupled multithreaded architecture that can be used in multi-core processor implementations. I also address how memory management functions can be off-loaded from processing pipelines to further improve system performance and eliminate cache pollution caused by runtime management functions.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Dr. Krishna Kavi, for his support, patience, and encouragement throughout my graduate studies. It is not often that one finds an advisor who always finds the time for listening to the little problems and roadblocks that unavoidably crop up in the course of performing research. I am deeply grateful to him for the long discussions that helped me sort out the technical details of my work. I am also thankful to him for encouraging the use of correct grammar and consistent notation in my writings and for carefully reading and commenting on countless revisions of this manuscript. His technical and editorial advice was essential to the completion of this dissertation.

My thanks also go to Dr. Saraju Mohanty and Dr. Phil Sweany, whose insightful comments and constructive criticisms at different stages of my research were thought-provoking and helped me focus my ideas.

I would like to thank Dr. Naghi Prasad and Mr. Satish Katiyar for their supports, kindly reminding and encouragements during the writing of this dissertation.

Finally, and most importantly, I would like to thank my wife Liqiong, my son Colin and my daughter Sydney. They form the backbone and origin of my happiness. Their love and encouragement was in the end what made this dissertation possible. I thank my parents, Yong and Li, my father-in-law Duanxiang for their faith in me and unconditional support.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Conventional superscalar architecture - based on the Von-Neumann execution model - has been on the main stage of commercial processors (Intel, AMD processors) for the past twenty years. The Von-Neumann architecture uses a program counter (PC) to control the execution of a program based on the control flow graph.

The superscalar architecture exploits the data independence at the instruction level and allows more than one instruction to be issued out-of-order in every clock cycle. The superscalar architecture dedicates a large amount of hardware to branch prediction, aggressive out-order-issue instructions, speculative instruction execution, etc., to get more instructions executed per cycle (IPC). It also utilizes a very deep pipeline to achieve a high clock rate. It has been shown that performance improvement diminishes by simply increasing the clock rate, or increasing the instruction scheduling window size with additional hardware resources in the superscalar architecture paradigm.

With the advances in VLSI technology, hundreds of billions of transistors can be packed into a single chip. With the increased hardware budget, how to take advantage of available hardware resources becomes an important research area. Some researchers have shifted from control flow Von-Neumann architecture back to dataflow architecture again in order to explore scalable architectures leading to multi-core systems with several hundreds of processing elements.

In this dissertation, I address how the performance of modern processing systems can be improved, while attempting to reduce hardware complexity and energy consumptions. My research described here tackles both central processing unit (CPU) performance and memory subsystem performance. More specifically I will describe my research related to the design of an innovative decoupled multithreaded architecture that can be used in multicore processor implementations. I also address how memory management functions can be off-loaded from

processing pipelines to further improve system performance and eliminate cache pollution caused by runtime management functions.

1.1. Dataflow Architecture

Dataflow architecture is a drastically different architecture from the current Von-Neumann architecture. An application can be translated into a data flow graph, which is a directed graph. The nodes in the data flow graph represent the instructions and the arcs in the data flow graph represent the data dependences between the instructions.

The execution of the dataflow architecture is only driven by the availability of the input operands. Once an instruction has all its inputs, it can be fired. The firing of an instruction will consume the input data (operands) and will generate the output operands. Since the conditions of an instruction that can be executed in the dataflow architecture are only dependent on data availability, dataflow architecture can easily exploit parallelism at the instructional level. Due to this fine-grained parallelism that can easily be exploited in dataflow architectures, the parallelism is much larger than that of the Von-Neumann model.

Dataflow architecture is a very powerful alternative to the current architecture. However, pure dataflow architecture, which requires a large amount of hardware such as the number of the functional units, communication bandwidth, and the operand matching hardware, is too complex to be implemented cost effectively.

To reduce the hardware requirement of pure dataflow architecture, some researchers proposed hybrid dataflow/Von-Neumann architecture. In such architectures, a dataflow graph is statically partitioned into subgraphs. These subgraphs are executed in a data flow order, but the instructions within subgraphs are executed sequentially. By doing this, the enormous scheduling and communication overheads are saved.

1.2. Scheduled Dataflow Architecture

Scheduled dataflow architecture (SDF) is a further enhancement of the hybrid dataflow and Von-Neumann architecture. The subgraphs of a data flow graph are transformed into threads. An application is partitioned into multiple threads with multiple contexts. A thread can be fired when all its inputs are available. Therefore, SDF exploits the thread level parallelism (TLP) inherent in the dataflow graph.

In modern processors, the CPU clock rate is much higher than the memory speed. The access times and I/O bandwidth did not improve with the processor clock rate. The gap between the processor speed and the memory speed has widened. One way to tolerate the long latency memory operation in modern computer systems is to perform a context switch. The hardware or the operating system swaps out the task that waits for the memory operation and schedules another ready task to run. Another approach to alleviate the memory gap is to use decoupled architecture. The main feature of decoupled architecture is separation of the memory access from the execution. In this type of architecture, there are two sets of different processing elements, one for the memory access and one for the execution. SDF architecture also incorporates this feature into its architecture design. SDF architecture also uses a non-blocking thread execution model to achieve a clean separation of memory access from execution.

This architecture differs from other multi-threaded architectures in two ways: 1) the programming paradigm is based on the dataflow, which eliminates the need for runtime scheduling, reducing the hardware complexity significantly and 2) complete decoupling of the memory access from the execution pipeline.

1.3. Thread-Level Speculation

With the advances in compiler technology in recent years, compiler-driven code parallelization has advanced significantly. Thread-level parallelism is the key to achieving high performance for today's multi-core architectures. Unfortunately, there are still a large set of applications that cannot be parallelized due to the inability to determine data dependences or control dependences at compiling time. These types of programs are very common in many application domains, like sparse matrix computations, molecular biology, image processing, etc. Furthermore, the loops in these applications are repeated for a large number of iterations with only a few data dependences across the loop iterations.

Thread-level speculation (TLS) is a technique which enables compilers to aggressively parallelize applications despite data or control dependences. In architectures with TLS support, the hardware takes the responsibility to check the dependence violations, and recover from the violations to guarantee the correct execution result. Due to the dataflow languages

are not commonly used, one has to compile the programs written in imperative languages like C/C++. These programming languages generally have no consideration of parallel dataflow programming paradigms. TLS support in SDF architecture will improve the compiler's ability to generate parallel threads, even with undetermined dependencies, to achieve a better execution performance.

1.4. Decoupled Memory Management Architecture

The second part of this dissertation research investigates efficient memory management needed in modern computing systems. Dynamic memory management is traditionally implemented as library functions within system's runtime library. The memory management functions execute in different logical slices of the applications. They rarely interfere with the applications except receiving parameters and returning values. But they compete for the cache and CPU resources, with the application. By off-loading the memory management functions from the CPU, the cache contentions can be reduced and the memory management performance will be improved.

1.5. My Contributions

In this dissertation, I propose a scalable thread-level speculation schema along with an architecture design for the scheduled dataflow architecture. This is the first work to propose a TLS schema in a hybrid dataflow architecture. Compared with the existing TLS schema, my design has the advantage of low hardware cost, and does not need any OS level support. I also model the performance of SDF architecture with TLS support using synthetic benchmarks. I compare the performance of TLS schema with the results reported by other researchers using selected benchmarks to show that thread-level speculation is more effective in SDF architecture. In my view, the beauty of the proposed TLS model and its hardware realization lies in its simplicity and clarity. I evaluated the performance impacts of hardware memory management in conventional architecture. I also proposed a hybrid hardware/software memory management implementation which can improve the performance of memory intensive applications by 23% with a small amount of hardware.

1.6. Outline of the Dissertation

The dissertation can be divided into two parts: the thread-level speculation (TLS) of SDF described in chapter 2 through chapter 5, and the hardware memory management described in chapter 6 through chapter 8. The dissertation is organized as follows: Chapter 2 will review SDF related dataflow architectures and architectures supporting thread level speculation; Chapter 3 will introduce SDF architecture in more detail; Chapter 4 will present my TLS schema; Chapter 5 will provide experimental results, analyses and comparisons with the results from other researchers; Chapter 6 will demonstrate the performance impacts of hardware memory management; Chapter 7 will compare the performance of different memory management algorithms; Chapter 8 will show a hybrid memory management unit design; and Chapter 9 will summarize contributions and some of the future improvements that can be applied to enhance the performance of my TLS schema.

CHAPTER 2

SURVEY OF RELATED WORKS

This survey includes two parts: the first refers to SDF-related dataflow architectures, and the second refers to thread-level speculation-related architecture (TLS).

2.1. SDF-Related Dataflow Architectures

SDF architecture has the following characteristics: it is a, multi-threaded hybrid dataflow/ Von-Neumann architecture, which decouples memory accesses and execution, and uses a non-block thread model. In this section, I will review works in traditional dataflow architecture, multi-threaded architecture, hybrid dataflow architecture, recent advances in dataflow architecture, decoupled architecture, and the non-blocking thread model.

2.1.1. Traditional Dataflow Architectures

The dataflow computing model was developed by Dennis and Misuanas[22] in the early 1970s. Instruction execution in dataflow architecture is based on the availability of input data. An instruction can be fired when all its inputs are available. Multiple instructions can be fired at the same time if sufficient resources are available. Dataflow architecture can easily achieve maximum instruction-level parallelism possible. The synchronization for parallel execution in the dataflow architecture is implicit (data availability). Single assignment languages and notions, such as SISAL[11], VAL[1] and Id[38], were developed during the same period. In conjunction with the dataflow architecture, anti-dependences (write after read) and output dependences (write after write) are handled elegantly. It seems very promising that this new architecture will deliver high parallelism machine. However, the architecture's performance as implemented is not as expected. This is partly due to the fact that VLSI technology was not mature at the time these architectures were implemented (in the 1970s and 1980s). Other limitations of the model include: 1) excessively fine-grained thread parallelism, 2) the fact that it's very hard to integrate high speed storage (like registers and caches) into this architecture, and 3) asynchronous triggering of instructions [47][48][70][72][66].

There are three main architectures based purely on the dataflow principle: Static (Single-Token-Per-Arc) Dataflow architecture, Dynamic (Tagged-Token) Dataflow architecture, and Explicit Token Store (ETS) architecture. The following sections describe some of these architectures.

2.1.1.1. Static Dataflow Architecture

In the static dataflow model, programs are represented as a collection of templates. A template contains the operation code, operand slots, and destination address fields. Destination address fields refer to the operand slots in subsequent templates that receive the results. Once the destination template receives a token, it needs to give an acknowledgment, which doubles the traffic. The instructions are fired when all the operands are available. Since only one set of operands can be held with each instruction template, the loop iterations can not be parallelized in this architecture. Architectures based on this model are: the MIT Static Dataflow Machine[22], the LAU in France, the DDMI at the University of Utah (1978), and the Texas Instrument DDP (1979).

2.1.1.2. Dynamic (Tagged-Token) Dataflow Architecture

Dynamic (Tagged-Token) dataflow architecture allows subprograms and loop iterations to proceed in parallel. A token consists of a tag and associated data. Each tag is composed of a PE number (to determine which processing element it refers to), context field t (uniquely identifying the context), initiation number i (identifying the loop iteration), and instruction address n. It also has a port number since the destination instruction may need more than one input. For example, a token can be defined as $<t.i.ni,\ data>p$. Port p represents left or right. If it is a simple token without identifying loop iteration, t is the tag itself. If the tags are attached to the tokens, they allow multiple tokens to reside on the arc and the architectures supporting such tokens are called dynamic (Tagged-Token) dataflow architectures. When identical tags are present on all input arcs (indicating a complete set of operands for a specific context and loop iteration), the node can be enabled or fired. There are two units, known as the matching unit and the execution unit, which are connected by an asynchronous pipeline with queues to balance the load variations. The execution is in terms of receiving, processing, and sending out tokens containing data and destination tags.

Data dependencies are translated into tag matching and transformation. The tag contains information about the destination context and how to transform the tag to achieve results. In order to support token matching, some form of associative memory, such as real memory with associative access, a simulated memory based on hashing, or a direct-matched memory, is required. The figure 2.1 below shows a block diagram, tag format, and token format of this particular architecture.

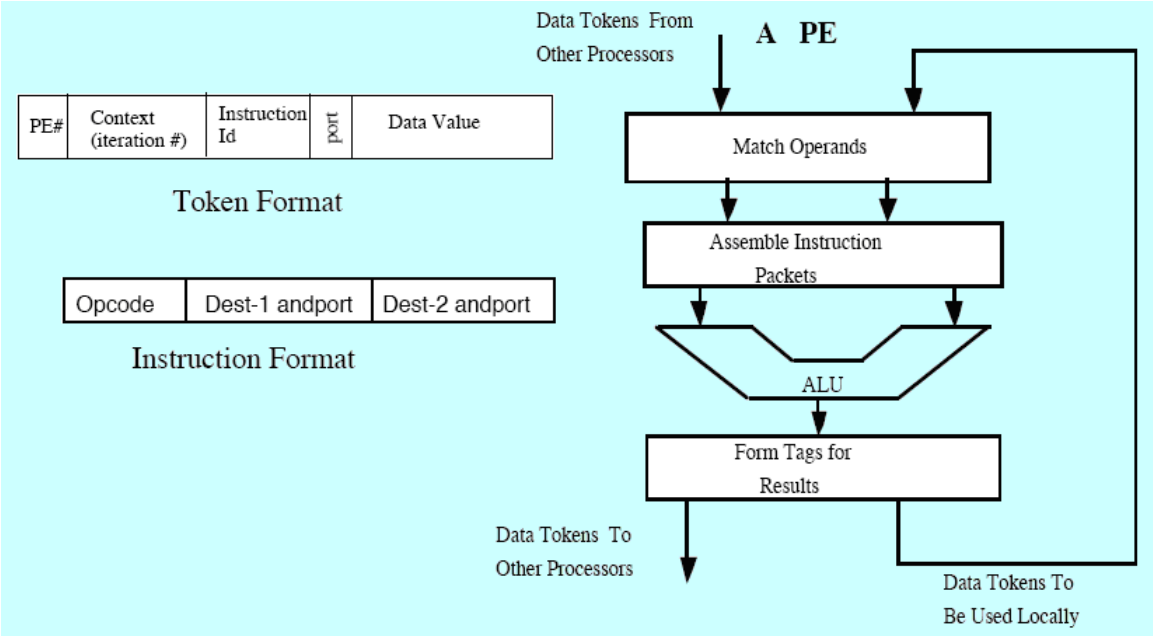


FIGURE 2.1. Block Diagram of Dynamic Dataflow Architecture

Dynamic dataflow unfolds much more parallelism than the static dataflow model. However, the large number of tokens waiting to be matched increases the size and complexity of the associative memory. Arvind et al. first proposed I-structure [8] [7], but it was simultaneously discovered by Watson and Gurd at the University of Manchester [77]. I-structure in dataflow can be viewed as a place to store structures, arrays, or indexed data using single-assignment rules. Each element of an I-structure can be read many times, but can only be written once. Each element is associated with status bits and a queue of deferred reads. The status can be defined as PRESENT (data ready), ABSENT (data not ready), and WAITING (data not ready, but with at least one deferred read). Three operations can be performed on the I-structures: I-allocate (allocation space for I-structure), I-fetch (to retrieve the contents; if the data is not ready, the read is deferred), and I-store (to write the contents). The I-fetch

instruction is executed in split phases, which means that the read request issued is independent in time from the response. It does not cause the issuing PE to wait for a response. The main drawback of this dynamic dataflow architecture is that the amount of memory needed to store tokens waiting for a match tends to be very large. To overcome this, the technique of hashing is used, which is not as fast as associative memory. Several dynamic dataflow machines have been constructed, including: 1) the MIT Tagged-Token Dataflow architecture [7], 2) the Manchester Dataflow Machine [36], 3) the Distributed Data-Driven Processor (DDDP) from OKI Electric Ind. (Japan) [49], 4) the Stateless Dataflow Architecture (SDFA) designed at the University of Manchester [23], and 5) the Data-Driven VLSI Array (DDA) designed by Technion (Haifa, Israel) [51].

2.1.1.3. Explicit Token Store Dataflow Architecture

The explicit token store (ETS) concept was proposed in order to overcome the drawbacks associated with token matching using associative memory or hashing. ETS uses direct matching of operands (or tokens) belonging to an instruction. In a direct matching schema, storage (called a frame) is dynamically allocated for all the tokens needed by the instructions in a code block. A code block can be viewed as a sequence of instructions comprising a loop body or a function. The actual disposition of locations within a frame is determined at compile time; however, the actual allocation of frames is determined at run time. In a direct matching schema, any computation is completely described by a pointer to an instruction (IP) and a pointer to a frame (FP). The pair of pointers, <FP, IP>, called a continuation, corresponds to the tag part of a token. A typical instruction that an IP points to specifies an opcode, an offset in the frame where the match of input operands for that instruction will take place, one or more displacements (destinations) that define the destination instructions to receive the result token(s), and the input port (left/right) indicator that specifies the appropriate input arc for a destination instruction.

When a token arrives at a node (e.g. ADD in Figure 2.2), the IP part of the tag points to the instruction that contains an offset r as well as displacement(s) for the destination instruction(s). The actual matching process is achieved by checking the slot availability in the frame memory at FP+r. If the slot is empty, the data value from the token is written

to the slot and its presence bit is set to indicate that the slot is full. If the slot is already full (indicating a match of input operands), the value is extracted, leaving the slot empty, and the corresponding instruction is executed. The resulting token(s) generated from the operation is communicated to the destination instruction(s) by updating the IP according to the displacement(s) encoded in the instruction (e.g., execution of the ADD operation produces two result tokens <FP.OP+1,3.55>. Instruction execution in ETS is asynchronous since an instruction is enabled immediately upon the arrival of the input operands.

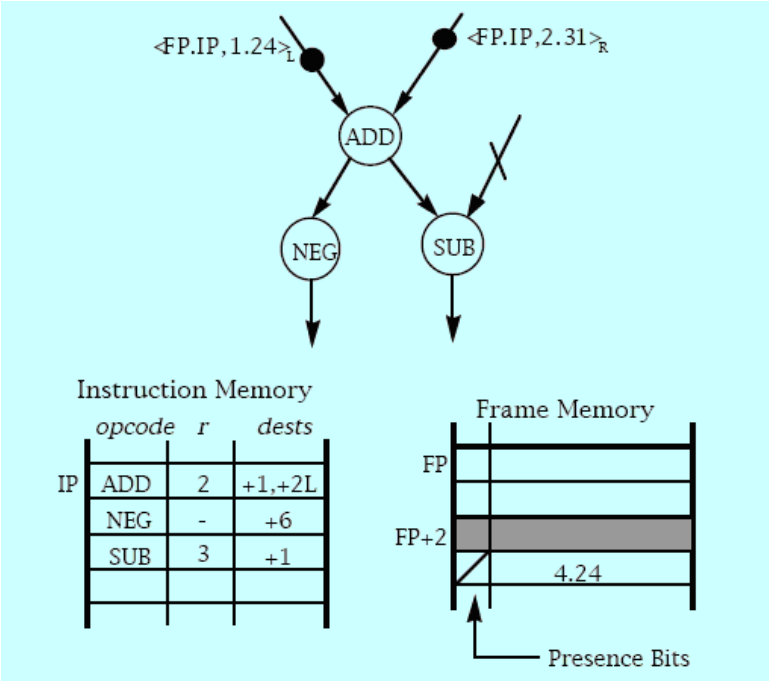


FIGURE 2.2. ETS Representation of a Dataflow Program Execution

MIT and Motorola jointly built the Monsoon explicit token store machine in 1990 [20][60]. The Monsoon PE used an 8-stage pipeline. The first stage is the instruction fetch. The second stage is the effective address generation, which consists of three pipeline stages. The execution stage consists of three pipeline stages, while the final stage comprises the token-form stage. The ETS model is also applied in other machines such as the EM-4 and Epsiln-2[34].

2.1.2. Hybrid Dataflow/Von Neumann Architecture

Even though dataflow architecture provided the natural elegance of eliminating output-dependencies and anti-dependencies, it performed poorly with sequential code. In an eight-stage pipeline machine such as the Monsoon, an instruction of the same thread can only be issued to the dataflow pipeline after the completion of its predecessor instruction. In addition, the token matching and waiting matching store introduced more pipeline bubbles or stalls into the execution stage(s) of the dataflow machines. In order to overcome these drawbacks, some researchers proposed a hybrid of dataflow/control-flow models along with a multithreaded execution. In such models [59][41], several tokens within a dataflow graph are grouped together as a thread to be executed sequentially under its own private program counter control, while activation and synchronization of threads are data-driven. Such hybrid architectures deviate from the original dataflow model, where the instructions fetch data from memory or registers, instead of having instructions deposit operands (tokens) in "operand receivers" of successor instructions. In such hybrid models two key features that support these types of architectures are: sequential scheduling and the use of registers for buffering the results between instructions. Examples of such hybrid models are: the threaded abstract machine (TAM) [19], P-RISC [59] and Star-T [4].

2.1.3. Recent Advances in Dataflow Architecture

Due to the drawbacks listed in the section 2.1.1, dataflow architectures did not reach commercial success. With advances in VLSI technology, the gap between the wire-delay relative to the switching speed and the exponential cost of the circuit complexity make a simple scaling up of existing processor designs futile [2]. Effectively translating the hardware budget into performance becomes a new challenge. Multithreaded model is becoming a preferred possible solution to this problem. Since dataflow model presents a clean model for multithreading, many researchers are shifting their interests back to dataflow architecture again.

There are two new architectures based on the dataflow concept: WaveScalar [69] architecture from the University of Washington and SDF[32] architecture.

2.1.3.1. WaveScalar Architecture

WaveScalar architecture integrates the idea of computation in memory (cache) and dataflow computing. The goal of WaveScalar is to minimize the communication cost between processors and memory. The key difference between WaveScalar and the previous dataflow architectures is that it supports traditional memory semantics.

WaveScalar instructions are cached and executed by an intelligent, distributed instruction cache - the WaveCache. The WaveCache loads instructions from the memory and assigns them to a processing element for execution. These instructions will reside in the WaveCache for multiple invocations. A WaveScalar binary is the data flow graph in executable format. It resides in the memory as a collection of intelligent instruction words.

The core of WaveScalar architecture is the WaveCache. A WaveCache is a grid of 2K (2048) processing elements (PEs) arranged into clusters of 16 PEs. Each PE contains the logic of the instruction placement and execution, input and output queues for instruction placement and execution, communication logic, and function units (integer and floating). Each PE can hold eight different instructions; a total of 16,000 instructions can be kept in cache. The instruction placement is dynamic based on the resource availability. And when an instruction is loaded, the destination field of an instruction is modified accordingly.

The WaveScalar compiler breaks the control flow graph of an application into waves. A wave has the following properties: 1) each instruction in a wave only executes once in a round, 2) the instructions are partially ordered, and 3) there is no control-flow within a wave. In WaveScalar, ϕ functions [18] are used to address the control dependences between waves. In this architecture, every data value carries a tag. The wave numbers are used to differentiate the data between dynamic waves. A special instruction, "WAVE_ADVANCE", is used to manage wave numbers. This architecture assumes the wave number computation is very simple and can be combined with other instructions, for example, the ADD-WITH-WAVEADVANCE. The key feature of WaveScalar is that wave number management can be entirely distributed and under software control, whereas tag generation in traditional dataflow architecture is only partially distributed.

Unlike traditional dataflow architecture, WaveScalar does not use I-structure memory. The memory ordering problem is solved by the compiler's insertion of the annotation <*pred, this, suc*> tuples into the *load* and *store* instructions. *Pred* is the sequence number of the predecessors, this is the sequence number of this node, and *suc* is the sequence number of the successor. In case of ambiguous dependences, the compiler will insert a '*?*' as a wildcard in the annotation.

The weakness of this architecture is that it moves the complex communication between instructions in traditional superscalar from the reservation stations to the PEs, and does not reduce the number of actual communications needed, as it claims: the architecture makes communications explicit between them. The other drawback is that the utilization of PEs (not functional units) is very low. The utilization of PEs is only 0.01% from the data provided [69]. One must conclude that today's technology cannot meet the requirements of WaveScalar.

2.1.3.2. SDF Architecture

SDF is one type of hybrid architecture that tries to use relatively simple hardware to achieve high performance, and to utilize the hardware more efficiently. The features of SDF architecture include decoupling and non-blocking multithreading. And SDF architecture is very flexible; one can choose the number of PEs according to different types of applications.

A detailed introduction to SDF architecture can be found in chapter three.

2.1.4. Multithreaded Architecture

There are several types of multithreaded architecture, which can be divided into three categories: multiple threads that share the same pipeline, multiple pipelines that share the same register files or L1 cache, and multi-core, chip-microprocessors (CMP).

2.1.4.1. Multiple Threads Using the Same Pipeline

This type of architecture has only one pipeline, but the hardware can explicitly support more than one thread (the hardware keeps multiple contexts). This kind of architecture falls into one of two categories depending on whether the instructions are issued from only a single thread or from multiple threads in a given cycle. In simultaneous multithreading (SMT) and

other interleaving architectures [74][3] instructions are issued from multiple threads to the pipeline during the same cycle.

Simultaneous multithreading (SMT) is an extension of mainstream superscalar architecture, which allows multiple independent threads to issue multiple instructions during each clock cycle to a superscalar pipeline. The Intel Hyper-Threading processor is an implementation of the SMT architecture. SMT simultaneously uses the thread-level parallelism and instruction-level parallelism by running multiple tasks on a superscalar processor at the same time. The entire active contexts in the SMT architecture will compete for resources at each clock cycle. The primary changes in SMT architecture from the conventional superscalar processor are the differences in the instruction fetch and register renaming mechanism. The fetch stage needs to fetch instructions from multiple instruction streams, which will incur extra overheads, so the fetch stage of SMT architecture has to be further divided into multiple stages. After the instructions are fetched from different contexts, the renaming register mechanism maps the architectural registers into the machine's physical registers. SMT architecture requires more physical registers than what a superscalar architecture does. Due to the large renaming register file size, the renaming of the register will take extra cycles. After register renaming, the instructions are combined into a single instruction stream in an instruction queue waiting for schedule. Because the different instruction streams do not have dependencies among them, it is possible to issue more instructions (higher ILP) in a single cycle. SMT improves the throughputs of the threads and utilizes hardware resources (functional units) more efficiently than superscalar. But SMT also increases the contention for resources including the branch predictor, and caches. Some studies show that this architecture is not very scalable in terms of the number of threads due to the contentions of these resources [39][40].

Tera utilizes a single pipeline by interleaving the execution of the instructions from different threads to hide the memory latency and the pipeline stalls due to data dependencies. The Tera computer does not include a cache. Each memory access takes 70 cycles but using the 128 hardware threads that can be supported, Tera aims to hide the memory latency since instructions from other threads are interleaved with the memory access instructions of a thread.

2.1.4.2. Multiple Pipelines

Multiple pipelines architecture has multiple pipelines layout on a single chip. Multiscalar [67], Superthreaded [73] architecture, and TRIPS [64] are examples of this architecture.

Multiscalar Architecture

A Multiscalar processor resembles multiple pipelines, but with a single logical register file that is implemented with physical copies in parallel processing units. In the Multiscalar architecture, the static program is partitioned into sequential "tasks". A task may be a basic block, a number of connected basic blocks, a number of loop iterations, a function call, or any combination thereof. The compiler is responsible for marking task boundaries, but it need not have full knowledge of inter-task or intra-task dependences. Dependences are implied by normal sequential execution semantics, as in conventional architectures. Load balance, i.e. choosing tasks of roughly equivalent size, is one of the important design considerations.

The control unit of the Multiscalar processor executes the program binary by processing each task as a single unit. Upon encountering each task, it predicts the subsequent task (by keeping a record of past history in a manner similar to dynamic branch prediction). Then the task is assigned to one of the parallel processing units. Consequently, the control unit scans the program (speculatively) by taking large steps, one task at a time, not pausing to look at any of the individual instructions within a task (including branches that may be contained within a task). The multiple processing units fetch and execute instructions from each of the assigned tasks in parallel. The result is that many instructions are executed per clock cycle.

Each task consumes and produces values that are bound to architectural registers and memory locations. Because of the natural sequential ordering of tasks, a value that a task consumes must be a value from an (earlier) task that produces it. Although copies are physically distributed, the register file has the appearance of a single, conventional file. When a register value is produced by one task and consumed by another, later task, the value must be conveyed to the physical register file in the later task (and all tasks in between). This is accomplished by the compiler and hardware. In a MultiScalar processor, tasks complete by committing their state changes (to registers and memory) in the same order in which they

were assigned to processors. This can only happen when the task has completed execution and when any predicted branch prior to the beginning of the task has been resolved.

Superthreaded Architecture

Superthreaded architecture has multiple processing elements (PEs) connected to each other in a unidirectional ring. Each processing element has a private instruction-level cache and a private memory buffer to cache speculative stores and to support run time data dependency checking. The PEs share the L-1 data cache and unified L-2 cache, as well as a register file and lock register.

This architectural model adopts a thread pipelining execution model that allows threads with data dependences and control dependences to be executed in pipelined fashion. The basic idea of thread pipelining is to compute and forward recurrence data and possible dependent store addresses to the next thread as soon as possible, so the next thread can start execution and perform run time data dependence checking. Thread pipelining also forces contiguous threads to perform their memory write-backs in order, which enables the compiler to fork threads with control speculation. With run time support for data dependence checking and control speculation, the Superthreaded architectural model can exploit loop-level parallelism from a broad range of applications.

TRIPS

TRIPS is a tile-based architecture. Each TRIPS core contains four processors. These processors share instruction caches and data caches, which are connected to processors through intelligent communication tiles.

Each processor contains 16 PEs (tiles). The PEs are connected in a grid fashion by an intelligent communication channel and share common register files. All the PEs contain an integer ALU and a floating point ALU, but each PE has it own reservation stations.

The compiler will partition the programs into blocks of the same size. A block is spawned as a thread for execution. Once a thread is scheduled, the instructions within the block will be placed over all the 16 PEs in a processor and the execution will be based on the dataflow model of that block.

TRIPs can be configured to support up to eight threads from the same task. The synchronization of the eight threads is through the register files and reservation stations, but these

threads must commit their results in the order of their spawning. TRIPS also can be configured to support four tasks with two threads from each task. By using dataflow execution within a thread, TRIP can utilize the fine-grained instruction-level parallelism (D-morph). And a processor can support multiple threads so that it also capitalizes on thread-level parallelism (T-morph). By the grid connection of the PEs with an intelligent communication channel, the data exchanges between neighboring PEs are greatly improved (S-morph). Because of these three properties, TRIPS is called a processor design with polymorphism for different applications.

2.1.4.3. Multi-Core Processors

When CMOS technology scales down in size, more and more transistors can be packed onto a single chip. Multiple independent cores, each representing a complete CPU, can be placed on a single chip with inter-core communication channels. Multi-core processors offer an immediate and cost-effective way to solve today's processor design challenge. There are several multi-core processors, like the Intel DUO and AMD Dual Core, which are already commercially available.

2.1.5. Decoupled Architecture

J.E. Smith [65] proposed a decoupled memory access and execution architecture that requires a compiler to explicitly slice the program into a memory access slice and a computational slice. The two slices would run on a different processing elements and a synchronization mechanism would be needed to guarantee the correct execution.

Rhamma [35] is a decoupled multi-threaded architecture that implements decoupled memory access and execution. Rhamma uses two separate pipelines: a memory access pipeline and an execution pipeline. A program interleaves on both pipelines during execution according to the instruction type. While a program running on the execution pipeline encounters a memory access instruction, a context switch will be generated and the program moved from the execution pipeline to the memory access pipeline. And when the memory access pipeline decodes a non-memory access instruction, a context switch will cause the thread to move back to an execution pipeline. The following figure 2.3 shows a block diagram of the Rhamma machine.

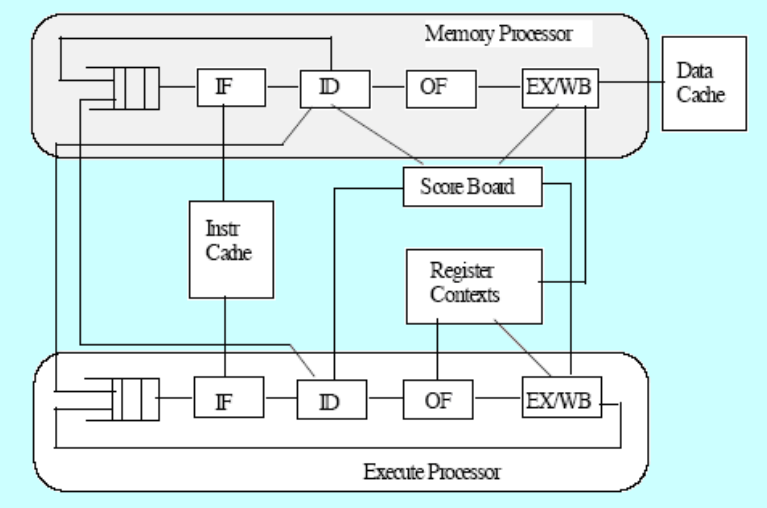


FIGURE 2.3. Block Diagram of Rhamma Processor

2.1.6. Non-Blocking Thread Model

A non-blocking thread proceeds to evaluation as soon as all input operands are available. It completes execution without blocking the processor due to the synchronization of threads. Thread context switching is controlled by the compiler with the means of generating new threads rather than blocking a thread for synchronization. It will reduce the synchronization overhead. The disadvantage of this model is that it tends to generate more fine-grained threads, which will in turn may increase the overhead.

The Cilk run-time environment [10] is a typical example of a non-blocking multi-threaded system.

2.2. Thread-Level Speculation

With the advances in compiler technology in recent years, compiler-driven code parallelization has advanced significantly. Thread-level parallelism is the key to achieving high performance for today's multi-threaded architectures. Unfortunately, there are still a large set of applications that cannot be parallelized due to the ambiguous data or control dependences. Those dependences are too complicated for the modern compiler to analyze. And those dependences are very common in many application domains, like sparse matrix computations, molecular biology, image processing, etc. Furthermore, the loops in these applications are repeated for a large number of iterations with only a few data dependences across the loop iterations in actual execution.

Thread-level speculation (TLS) is a technique which enables compilers to aggressively parallelize applications despite data or control dependences between the resulting thread. In architectures with TLS support the hardware assumes responsibility of checking dependence violations and recovering from the violations to guarantee the correct execution result.

TLS support can be partitioned into three categories according to the architecture: TLS support on a single chip, TLS support on a distributed system, and TLS support on both a single chip and distributed system.

2.2.1. Single Chip TLS Support

Marcuello et. al. [57] proposed a multi-threaded micro-architecture that supports speculative thread execution within a single processor. The micro-architecture of a speculative multi-threaded processor (SM) consists of several thread units (TU) that execute the different threads of a program in parallel. TUs are connected using ring topology.

Each thread unit has its own physical registers, register map table, instruction queue, functional units, local memory, and reorder buffer. The loops are captured by the hardware to automatically generate parallel threads. Control speculation is used in two levels, one for the parallel loop generation, and the other for the branches. Initially, there are no speculative threads. When the non-speculative threads start a new iteration of a loop, a number of speculative threads are created and allocated to execute the subsequent iterations. When a speculative thread reaches the end of its iteration, it is suspended and waits to commit or be squashed. After the non-speculative thread is committed, the speculative thread of the next iteration either becomes non-speculative or squashed.

Data dependences across iterations can be categorized as either register dependences or memory dependences. The register dependences are solved by a loop iteration table, which records the detailed value changes of a register in that iteration. The memory dependences are solved by a multi-value cache. For each address, the multi-value cache holds the iteration number, data value, and a flag indicating whether the data value has been produced. The data is loaded or stored by that iteration. If the earlier iteration updates an address in the multi-value cache that is labeled and read by a later iteration, the multi-value cache will generate a dependency violation and the later iteration will be squashed and re-executed.

Multiscalar uses a centralized ARB (Address Resolution Buffer) [31], which is similar to a multi-value cache to support thread-level speculation. Each load will update the ARB states and each store will check the buffer to guarantee that there are no memory dependency violations while a thread commits. Because the multi-value cache and ARB are centralized, they may become the bottleneck of the system. Speculative versioning cache [33] is an attempt to distribute the centralized ARB to each processing unit in order to avoid the delay caused by serializing the operations on the ARB. The speculative versioning cache provides a private cache for each processing unit. The system is similarly organized to a snooping bus-based cache coherent symmetric multiprocessor (SMP). The architectural organization is as Figure 2.4:

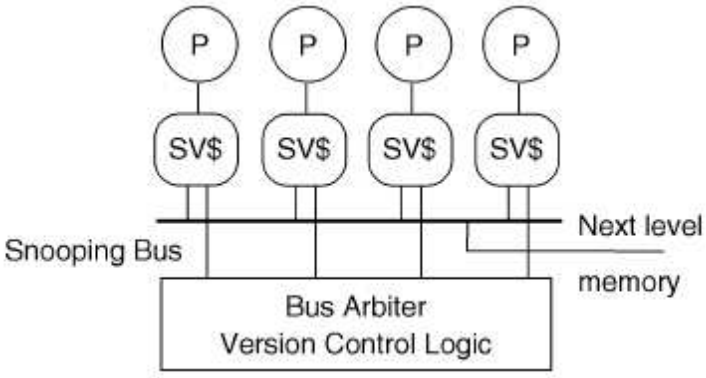


FIGURE 2.4. Architecture for SVC

There are two additional fields added to each speculative versioning cache line. One is the L bit; the other is the pointer field. The L bit is set when a task loads from a line before storing it to the line. If the L bit is set it means that there is a potential memory dependency violation. The pointer field is used to store the L1 cache that contains the next copy, which forms a distributed versioning ordering list (VOL). The requested hit on the private cache will not generate an additional request to the version control logic. The cache misses will issue a bus request, which is snooped by all the caches, and the version control logic will use the information from the VOL to form the appropriate response. When a task commits, all dirty lines in its private cache will write back to the memory and other lines are invalidated. When a task is squashed, all the lines in the cache are invalidated. The cache coherence schema used in speculative versioning cache is an extension of an invalidate-based

write-back cache coherence protocol. Instead of running this protocol on an SMP system, it runs between tightly coupled PEs.

Stanford Hydra [37] is a CMP system which supports the TLS execution. The CMP contains four MIPS processors. Figure 2.5 shows the design of Hydra.

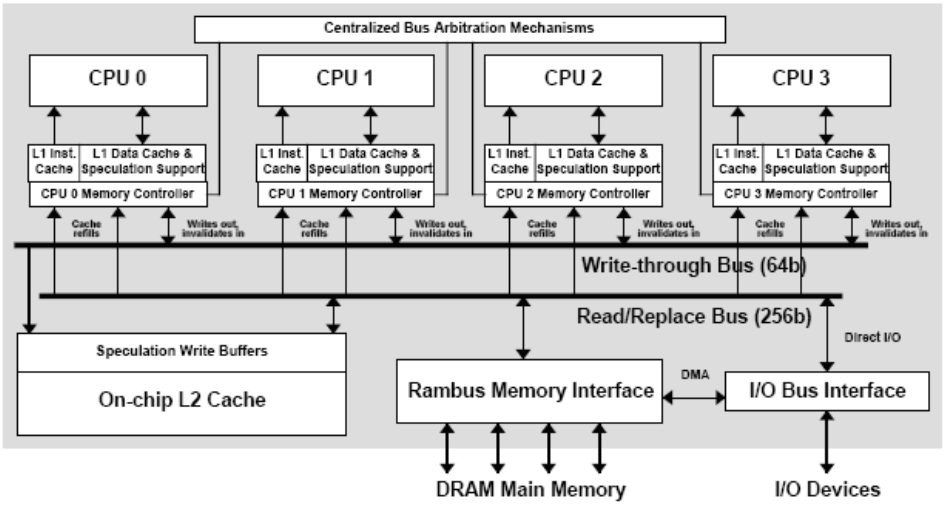


FIGURE 2.5. Block Diagram of Stanford-Hydra Architecture

Each of these four CPUs has a speculative coprocessor. The coprocessor will set up the speculative context and copy the register file to a Register Passing Buffer. This architecture also utilizes a variation of the invalidate-based write-back cache coherence protocol to detect data cache access violations. There are two sets of bits with each L1 data cache line to support the TLS execution.

The first set of bits includes a modified bit and a pre-invalid bit for invalidating a cache line. The modified bit is used to buffer the speculative memory state. By using the write-back protocol, the modified data will also be put on the bus with its sequence number. Only the CPUs that are running speculative threads will update its local cache. Once the pre-invalid bit is set, it means a speculative thread has modified this line. This bit is used to delay invalidating this cache line until a speculative thread has been assigned to the CPU. The second set of bits includes a read bit and a write bit used to detect the data dependency violation of the speculative threads. These two bits are designed to allow gang-clearing(invalid all the cache line accessed by this speculative thread in once) when a thread is either squashed or restarted. A read bit set means the thread has read this data. Once a

write from a less speculative thread is seen on the bus, the cache will generate a dependency violation and it will cause the speculative thread to be squashed. This bit set means that the thread has generated a local version of the address referenced.

Krishnan et al. [52] proposed a schema that supports the TLS execution using an memory disambiguation table (MDT) to solve the ambiguous memory dependences. The MDT is similar to a directory in a conventional shared-memory multiprocessor system. Figure 2.6 shows an example of MDT. The assumption is that each processor has a private L1 cache and a shared L2 cache. The MDT is integrated with the shared L2 cache. MDT keeps entries on a per memory-line basis and maintains information per word basis. For each word, the MDT keeps a load bit and a store bit for each processor. When a thread is initialized on a processor, all its LOAD bits and STORE bits are cleared. As the thread executes, the MDT works like a directory that keeps tracking which processor shared which words. By doing this, MDT can find the violation of the memory accesses.

Valid	Address Tag	Load Bits (Word 0) $L_0L_1L_2L_3$	Store Bits (Word 0) $S_0S_1S_2S_3$
1	0x1234	0 0 1 0	0 1 0 0
0	0x0000		
...			
1	0x4321	0 0 1 1	0 1 0 0

FIGURE 2.6. Example of MDT

2.2.2. TLS Support for the Distributed Shared Memory System

Zhang et al. [81] proposed a schema that supports speculative thread execution in large-scale distributed shared memory (DSM) systems by extending the directory-based cache coherence protocol. This schema partition the loop iterations into read-first iteration or writing iteration based on the non-privatization, blocked privatization, advanced privatization, and blocked advanced privatization algorithm in compile time. The directory controller will create a undo log of the memory accessed by these threads. Once the directory-controller detects a memory access violation, it will invalid the speculative threads and recover the correct memory state according to the undo log. These schema requires the speculative thread to commit in order of their iteration number.

2.2.3. Scalable TLS Schemas

Cintra et al. [16] proposed a TLS schema that supports CMP and a large-scale DSM system. This schema extends the MDT schema for the CMP, described in section 2.2.1. For each node of CMP, there is a local MDT (LMDT) table, the same as the MDT in [52]. For the whole DSM system, there is a global memory disambiguation table (GMDT). The GMDT is distributed to all the nodes in the system as the directory. GMDT keeps the memory information on a per cache-line basis instead of a per word basis, such as in the MDT. In this architecture the threads assigned to each node must be in batch mode (or a chunk), so the GMDT can view the chunk of threads on a node as one speculative job, thus the design of GMDT can be greatly simplified. The GMDT uses a ring structure to perform the chunk-to-node mapping. As the driving force of the execution, at least one chunk is non-speculative in the system.

Instead of squashing one thread as in the Zhang et al.[81], the GMDT squashes a chunk of threads at one time. All the GMDTs synchronize in a barrier to ensure the integrity of the system.

Steffan et al. [68] proposed a TLS architecture that is built upon a write-back invalidation-based cache coherence protocol. This design uses the single-chip multiprocessor or simultaneous multi-threaded processors as the building block. The base architecture is as Figure 2.7:

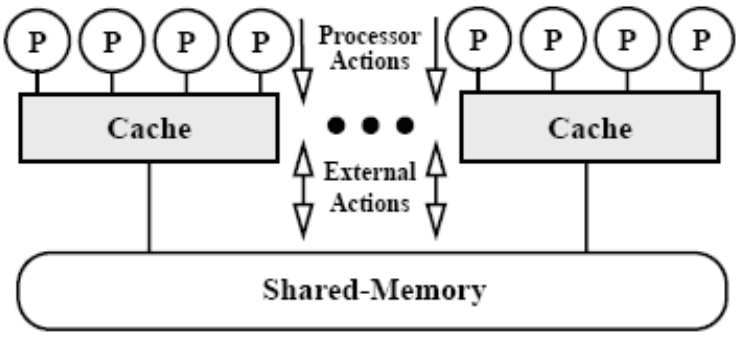


FIGURE 2.7. Block Diagram of Architecture Supported by Steffan's TLS Schema

This schema depends on the speculative threads committing based on their logical or program order. An epoch number is used to define the logical order of the threads. It adds the

speculative-exclusive (SpE) state to define the speculative-modified data, and speculative-shared state to define the speculatively read data. To define a speculative thread context, the following resources are needed: epoch number, cancel handler address, violation handler address, violation flag, logic late mask, replication of the SL and SM bits from cache, and an ownership buffer (ORB), which is used to record the address of the data that was speculatively modified by this line. The paper does not mention how the thread context is maintained. It can be maintained by hardware, but how to implement this hardware with very low overhead will be another challenge. Or if it is maintained by an operating system, this will result in very slow memory access for speculative threads.

This work is most closely related to ours. The difference between my work and Steffan's is that his work is based on conventional architecture, while my work is based on dataflow/Van Neumann hybrid architecture. If provide a set of fully implementable hardware design that supports thread-level speculation in SDF architecture. This design has a very low hardware complexity. All the speculative stores will be buffered in the registers instead of in the ORB, as they do in their architecture. This hardware does not need any help from operating systems.

CHAPTER 3

SDF ARCHITECTURE OVERVIEW

SDF architecture applies the dataflow model on a thread-level granularity, instead of on the fine-grained instruction-level like conventional dataflow architectures. SDF architecture is a decoupled architecture that separates memory access from the main execution pipeline to further tolerate the latency of memory access by overlapping the memory access with the execution. SDF architecture also utilizes the non-blocking thread model to reduce the context switching cost.

3.1. SDF Instruction Format

Unlike conventional dataflow architecture, SDF instructions within a thread are executed in a control-flow manner with a private program counter that is associated with each thread. The instruction format of SDF architecture is very similar to the MIPS instruction format. Most instructions in SDF have two source operands and one destination operand like MIPS instructions. Due to the fact that SDF architecture maintains I-structure semantics for thread synchronization, and conventional memory semantics like MIPS, SDF has two sets of memory access instructions for memory accesses – IFetch and IStore for I-structure memory, and Read and Write for conventional memory.

SDF also has some special instructions for spawnning threads (FALLOC), thread switching between the memory access and execution pipelines (FORKSP, FORKEP), and reading and writing from the frame memory (Load and Store).

3.2. SDF Thread Execution Stages

For SDF architecture, the compiler partitions the applications written in high-level languages into threads based on the data flow graph. A thread in SDF architecture can be viewed as a subgraph of a data flow graph. The size of an SDF thread is limited by the hardware resources, such as number of registers per context, number of entries in a frame memory for a thread, etc. Because SDF decouples memory access and execution, a thread

will further partition into three portions: pre-load codes, execute codes, and post-store codes. The pre-load and post-store codes will execute on the memory access pipeline, and the execution code will execute on the execution pipeline. The execution sequence of a thread is shown in the following figure 3.1.

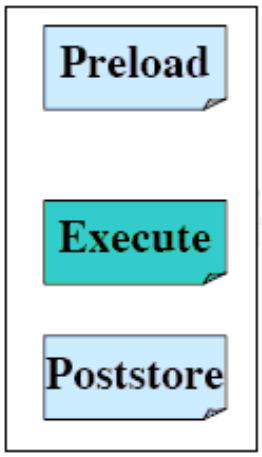


FIGURE 3.1. Code Partition in SDF Architecture

To understand the decoupled, scheduled dataflow concept, consider one iteration of the innermost loop of matrix multiplication: $c[i,j] = c[i,j] + a[i,k] * b[k,j]$. The SDF code is shown in figure 3.2. In this example, I assume that all necessary base addresses and indexes for the arrays are stored in the thread's frame. The thread is enabled after it receives all inputs in its frame and a register context is allocated. In the preload portion of the example, the base address and offsets of arrays are loaded from the frame memory and then the data element referenced in this iteration is fetched from the I-structure memory. By executing the FORKEP instruction, the thread gives up the memory access unit and the continuation of the thread is put into a queue to wait for the execution unit. When the thread is scheduled on the execution pipeline, the multiplication is performed. Then by executing the FORKSP instruction, the continuation of this thread is put into the post-store queue to wait for the execution. Finally, the thread gets the memory access unit again and the result is post-stored. Because the thread model is non-blocking, the FFREE instruction frees the frame held by the thread and the continuation is destroyed.

```
Preload:    LOAD     RFP|2, R2       # base address of a into R2
            LOAD     RFP|3, R3       # index a[i,k] into R3
            LOAD     RFP|4, R4       # base address of b into R4
            LOAD     RFP|5, R5       # index b[k,j] into R5
            LOAD     RFP|6, R6       # base address of c into R6
            LOAD     RFP|7, R7       # index c[i,j] into R7
            IFETCH   R2, R3, R8      # fetch a[i,k] to R8
            IFETCH   R4, R5, R9      # fetch b[k,j] to R9
            IFETCH   R6, R7, R10     # fetch c[i,j] to R10
            FORKEP   Body            # transfer to EP
            STOP

Body:       MULTD    R8, R9, R11     #a[i,k]*b[k,j] in R11
            ADDD     R10, R11, R10   # c[i,j] + a[i,k]*b[k,j] in R10
            FORKSP   Poststore       #transfer to SP
            STOP

Poststore:  ISTORE   R6, R7, R10     #save c[i,j]
            FFREE                    #free the frame
            STOP
```

FIGURE 3.2. An SDF Code Example

3.3. Thread Representation in SDF

A thread in SDF architecture can be uniquely identified by a continuation of four tuples - <FP, IP, RS, SC>. FP is the Frame Pointer (where thread input values are stored), IP is the Instruction Pointer (which points to the thread instructions), RS is a register set (a dynamically allocated register context), and SC is the synchronization count (the number of inputs needed to enable the thread). The synchronization count is decreased when a thread receives its inputs, and the thread is scheduled on SP when the count reaches zero. Each thread has an associated continuation. At any given time a thread continuation can be in one of the following states:

- Waiting continuation (WTC) or <FP, IP, −−, SC>
- Pre-Load Continuation (PLC) or <FP, IP, RS, −−>
- Enabled Continuation (EXC) or <−−, IP, RS, −−>
- Post-store Continuation (PSC) or <−−, IP, RS, −−>

The "−−" means that the value is undefined in that state.

Figure3.3 shows a state transition diagram of thread continuation. A thread continuation is created in the WTC state. A thread moves from the WTC state to the PLC state once a register set is assigned to the continuation and scheduled to an SP (memory access unit)

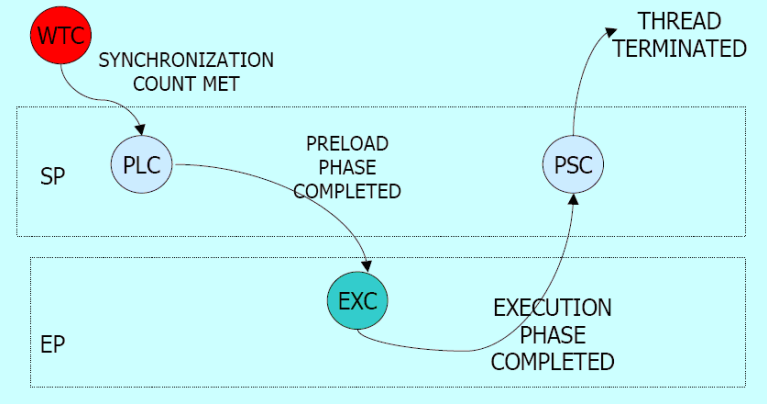


FIGURE 3.3. State Transition of the Thread Continuation

for preload. And then the thread is moved to EXC state, when it finishes using SP and transfers to EP (execution unit). After using EP, the continuation is moved to SP again and the thread continuation enters PSC state. The scheduling unit is responsible for moving continuations between SP and EP.

3.4. The Basic Processing Units in SDF

A processing element in scheduled dataflow architecture (SDF) is composed of three components: the Synchronization Processor (SP), the Execution Processor (EP), and the thread schedule unit.

The SP is responsible for pre-loading a thread context (i.e. registers) with data from the thread's memory and post-storing results from a completed thread into frames of destination threads. The synchronization pipeline consists of six stages: instruction fetch, instruction decode, effective address computation, memory access, execution, and write-back. Figure 3.4 shows a block diagram of SP. The instruction fetch stage fetches an instruction belonging to the current thread using the PC (Program Counter). The decode stage decodes the instruction and fetches the operands from the registers. The effective address stage computes the effective address for memory access instructions like LOAD, STORE, READ, WRITE, IFETCH, and ISTORE. The memory access stage completes the memory access for memory access instructions. The write-back stage completes the LOAD, READ, and IFETCH instructions by storing the result into the register. Unlike traditional dataflow architecture, the instruction is not scheduled immediately when the operands are available. The instructions

are "scheduled" like control flow architectures using program counters. The instruction-driven approach eliminates the need for complex communications to exchange tokens among processing elements.

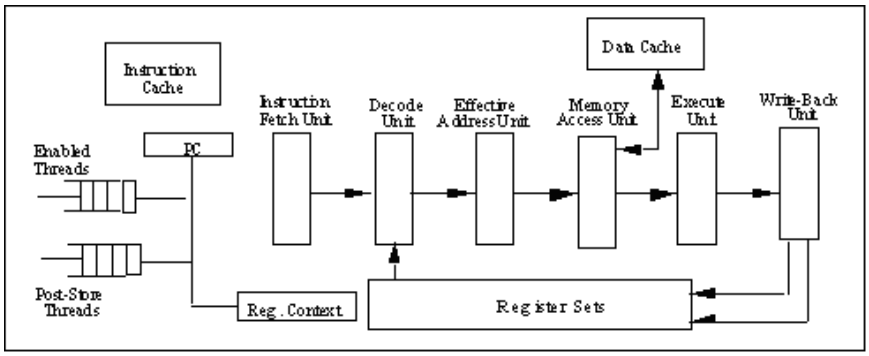


FIGURE 3.4. SP Pipeline

The EP performs thread computations, including integer and floating point arithmetic operations, and spawns new threads. The execution pipeline consists of four pipeline stages: instruction fetch, decode, execute, and write-back. The instruction fetch stage behaves like a traditional fetch unit, which fetches instructions pointed by the program counter (PC). The instruction decode stage decodes the instruction and fetches the operands from the registers. The execution stage executes the instruction, and the write-back units write the value to the register file.

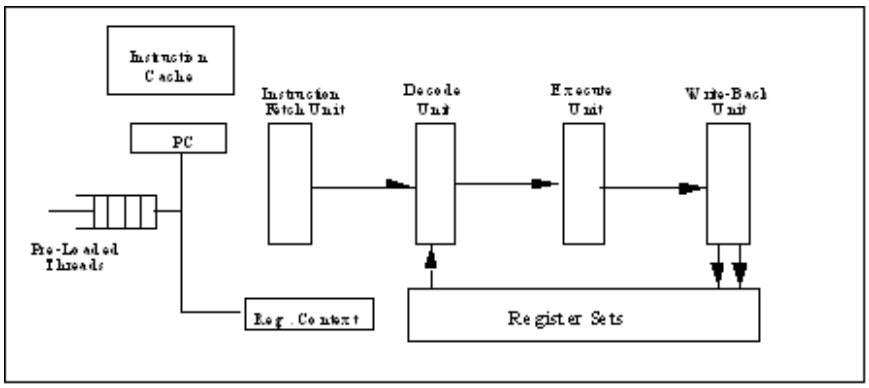


FIGURE 3.5. General Organization of the Execution Pipeline (EP)

A more general implementation can include multiple EP and SPs to execute threads from a single task or independent tasks. Multiple SP and EPs can be configured into multiple clusters. Inter-cluster communications will be achieved through shared memory.

The scheduling of threads to an SP and EP is handled by a scheduling unit (SU). Figure 3.6 shows the SP with scheduling queues.

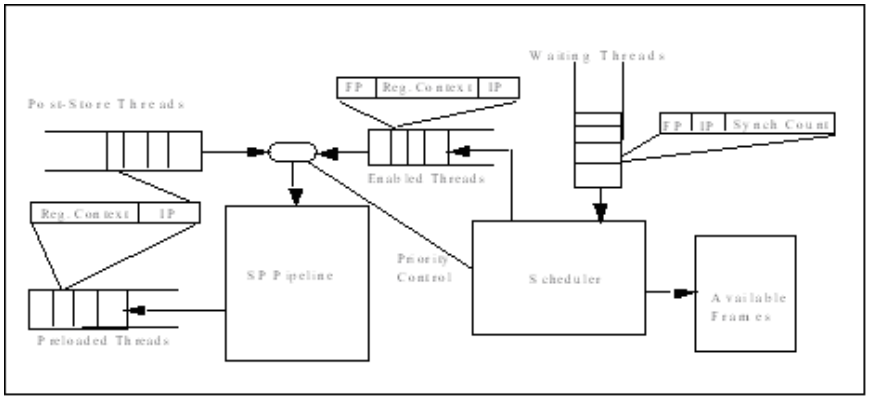


FIGURE 3.6. Overall Organization of the Scheduling Unit

In SDF architecture, a thread is created by using the FALLOC instruction. The FALLOC instruction creates a frame (pointed to by an FP) related to a certain thread (pointed to by an EP) with a given Synchronization Count (SC), which indicates the number of inputs needed to enable the thread. The thread continuation (<FP, IP, −−, SC>) is then handled by the SU. The SU takes care of checking when the synchronization count reaches zero. Then it allocates a register set (RS) to it, and the continuation is scheduled for execution on the SP. Now the thread is in the PLC state. The PLC thread is ready to execute on the SP for preload. At the end of the preload phase, the thread is handed off to the EP by executing the FORKEP instruction. The thread is in state EXC (<−−, IP, RS, −−>) and is ready to be executed on the EP. The SU is also responsible for moving the thread between the SP and the EP. The FORKSP instruction is used to move the thread from the EP to the SP (from the PLC state to the PSC state). After completing the post store, the register set (RS) is freed and the thread execution is complete.

In order to speed up the frame allocation, a fixed-size frame for the thread is pre-allocated and a stack of indexes pointing to the available frames is maintained. The SU makes a frame index available to the EP by popping the stack when executing the FALLOC instruction. The SU is also responsible for allocating the register sets. The register sets are viewed as circular buffers which are allocated to enabled threads. The SU pushes indexes of de-allocated frames to the stack FFREE instruction subsequent to the post-store of completed

threads. These policies permit fast context switching and low overhead creation of threads. FALLOC and FFREE take two cycles in SDF architecture. Because the thread transition in between SP and EP are very simple, I assume these transitions can be completed in two cycles based on Rhamma architecture [35]. Note that scheduling is at the thread-level rather than instruction-level, unlike superscalar or other conventional architectures, and thus it requires much simpler hardware.

3.5. Storage in SDF Architecture

There are three classes of storage in SDF architecture: I-structure, conventional memory, and frame memory. I-structure follows the semantics of I-structure proposed by Arvind [8][7]. To access this memory IFETCH and ISTORE instructions are used. The IALLOC instruction is used to allocate the I-structure and the IFREE instruction is used to de-allocate the I-structure. The I-structure can be used to model the producer and consumer problem and will synchronize the thread execution.

The frame memory is used to store the inputs from one thread to the continuation of another thread. In addition, the frame memory is allocated by using the FALLOC instruction and de-allocated by using the FFREE instruction. Allocation and deallocation of frames are the EP's responsibility.

The conventional memory is accessed by using the READ and WRITE instruction. It is just like the memory in any Von Neumann architecture.

3.6. The Experiment Results of SDF Architecture

The SDF architecture exploits two levels of parallelism. Multiple threads can be active simultaneously, permitting thread-level parallelism. In addition, the three phases of a thread (pre-load, execute, and post-store) can be overlapped with those of other threads. I reported the results of preliminary performance comparisons of SDF with other architectures [32][46]. More recently, If also have collected data to compare SDF with SMT. The results thus far show that SDF scales better than superscalar, and VLIW systems with added functional units and register sets. Even for control-dominated applications, multi-threading computations from independent workloads can lead to better utilization of the functional units within SDF (similar to SMT). Here I show a selected subset of my results.

Table 3.6 compares SDF with VLIW (using Trimaran [17]) and superscalar (using Simplescalar [13]) for a selected set of benchmarks (Matrix Multiplication, a picture zooming program [50], JPEG, ADPCM from EEMBC), in terms of the instructions per cycle (IPC). It should be noted that SDF uses in-order execution pipelines and performs no speculative executions. Thus, the actual (useful) number of instructions executed by SDF is much smaller than those in the other architectures. In these comparisons, I set the number of SPs equal to integer functional units of VLIW and superscalar systems, while setting the number of EPs equal to the number of floating point functional units. Since I rely on in-order execution, I feel that this is a reasonable comparison. EPs are provided with a full complement of floating point arithmetic units, while SPs are designed with simple integer add/multiply units to compute memory addresses.

The following table clearly shows the scalability of SDF as more functional units are added[1], reaching 2.7 instructions per cycle with four EPs and four SPs for matrix multiplication.

Table 3.6 compares SDF with SMT[2] [56] (in terms of IPC), using the same number of thread contexts and functional units. This data indicates that SDF can perform as well as SMT.

The SDF code is generated by a toy compiler, which can only compile simple C programs. The toy compiler is built upon the SUIF compiler tool[3]. The data supports the hypothesis that one can design architectures that are based on a fine-grained multi-threading model that scales better than and compares favorably with conventional modern architectures (at least for the selected benchmarks). More sophisticated compiler optimizations, and optimizations that uniquely take advantage of this decoupled architecture, will certainly improve the performance of SDF over competing architectures.

[1]Superscalar architecture may improve its IPC if larger instruction windows, renaming registers, and reservation stations are provided.

[2]I used the simulator available at http://maggini.eng.umd.edu/vortex/ssmt.html. I acknowledge that IPC may not be the best measure, but since the SMT simulator available to us only provided IPC counts, I included only IPC comparisons in my tables. The comparisons are also limited by the SMT simulator, which could not run large benchmarks or run benchmarks with a large number of thread contexts.

[3]SUIF group, `http://suif.stanford.edu`

	IPC VLIW	IPC Superscalar	IPC SDF
Benchmark	1 IALU/1 FALU	1 IALU/1 FALU	1 SP / 1 EP
Matrix Mult	0.334	0.825	1.002
Zoom	0.467	0.752	0.878
Jpeg	0.345	0.759	1.032
ADPCM	0.788	0.624	0.964
Benchmark	2 IALU / 2FALU	2 IALU / 2FALU	2 SP / 2 EP
Matrix Mult	0.3372	0.8253	1.8244
Zoom	0.4673	0.7521	1.4717
Jpeg	0.3445	0.7593	1.515
ADPCM	0.7885	0.6245	1.1643
Benchmark	4 IALU / 4FALU	4IALU / 4FALU	4 SP / 4EP
Matrix Mult	0.3372	0.826	2.763
Zoom	0.4773	0.8459	2.0003
Jpeg	0.3544	0.7595	1.4499
ADPCM	0.7885	0.6335	1.1935

TABLE 3.1. SDF versus VLIW and Superscalar

	IPC SMT	IPC SDF
Benchmark	2 threads	2 threads
Matrix Mult	1.9885	1.8586
Zoom	1.8067	1.7689
Jpeg	1.9803	2.1063
ADPCM	1.316	1.9792
Benchmark	4 threads	4 threads
Matrix Mult	3.6153	3.6711
Zoom	2.513	2.9585
Jpeg	3.6219	3.8641
ADPCM	1.982	2.5065
Benchmark		6 threads
Matrix Mult		5.1445
Zoom		4.223
Jpeg		4.7495
ADPCM		3.7397

TABLE 3.2. SDF versus SMT

CHAPTER 4

THREAD-LEVEL SPECULATION (TLS) SCHEMA FOR SDF ARCHITECTURE

The TLS schema is based on a variation of the invalidate-based snoopy cache coherency protocol, thus I will briefly review the cache coherency protocol at the beginning of this chapter. Then I will introduce our TLS schema.

4.1. Cache Coherency Protocol

Cache coherency problems arise in the multiprocessor systems, where each processor has its own private cache and multiple caches can have copies of the same memory location simultaneously. Cache coherency protocol is the mechanism that ensures all these copies remain consistent when the contents of that memory location are modified. There are three main types of cache coherency mechanisms: snooping cache coherency protocol, directory-based cache coherency protocol, and compiler-directed cache coherency.

Snoopy cache protocols are based on the systems where all the processors in the system and the memory module share a common bus. All the processors monitor the transactions on the shared memory by snooping on the bus. There are several variations of the snoopy cache protocol: MSI [30], MOSI [45], MESI [61] [58], and MOESI.

In these cache coherence protocols, each cache line in the system is in one of the predefined states. The most common states in these protocols are as follows: Modified (M), Invalid (I), O (Owned), S (Shared), and E (Exclusive). Every cache controller observes every write on the bus. If a snooping cache has a copy of the block, it either invalidates or updates its copy. Protocols that invalidate cached copies on a write are commonly referred to as invalidation-based protocols, whereas those that update other cached copies are called update-based protocols. In either case, the next time the processor with the copy accesses the block, it will see the most recent value, either through a cache miss or because the updated value is in its cache. Archibald et al. [5] reviewed these cache coherence protocols and the performance

issues of snooping cache protocol. The snooping cache protocol is often used in symmetric multiprocessor (SMP) systems.

With a directory-based cache coherent protocol [71] [79], a processor must communicate with a common directory whenever the processor's action may cause an inconsistency between its cache and the other caches or memory. The directory maintains information about which processor has a copy of which block. Directories usually track where data is located in a multiprocessor at the granularity of a cache block. Every request for data (i.e. every "read miss") is sent to the directory, which in turn forwards information to the nodes that have cached that data. A directory-based cache coherency protocol is often used in distributed memory systems such as the non-uniform memory access system (NUMA).

Both the directory-based cache coherency protocol and snoopy cache protocol can be enforced by a cache controller.

The compiler-directed [75] [12] [27] cache coherence mechanism determines, at compile time, which cache block may become stale and inserts special instructions into the code to ensure the stale cache block will not be used.

In order to correctly execute an application, the TLS schema must detect the memory access violations, where a cache coherency protocol can be applied for this purpose. Our underlying system is an SMP, so in this work I extend a variation of the invalidate-based snoopy cache protocol to support our TLS execution.

4.2. The Architecture Supported by the TLS Schema

For non-speculative SDF architecture, if there is an ambiguous RAW (true dependence) that cannot be resolved at compile time, the compiler generates sequential threads to guarantee correct execution using I-structure semantics. This will reduce the performance of certain programs. However, with hardware support for speculative thread execution and result commit, a complier can more aggressively create concurrent threads to improve the performance of applications.

The purpose of TLS schema is to support speculative execution both within a cluster (which includes multiple SPs and EPs) and multiple clusters connected by a shared bus with a memory unit. Because our architecture has the property of decoupling memory access,

that means only the memory access pipeline (SP) will access the memory. The data cache is connected only to the SPs. Figure 4.1 shows the architecture supported by our TLS schema.

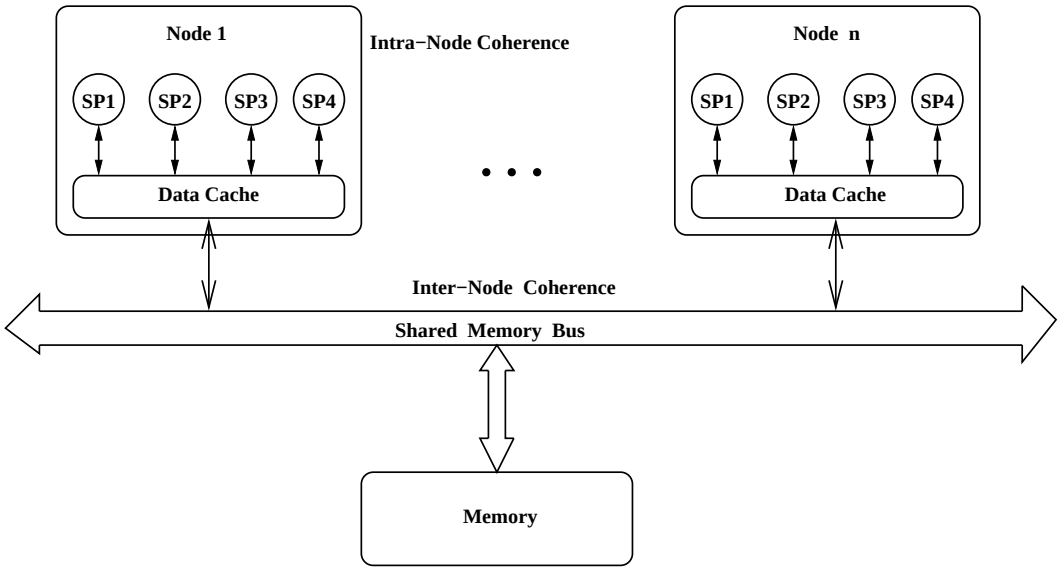


FIGURE 4.1. Architecture Supported by the TLS schema

Since EPs do not access the memory, EPs are not included in this block diagram. In each node, the SPs share the same data cache. TLS execution must maintain coherence within the same node and across multiple nodes.

The TLS schema is scalable both within a node, which means the number of speculative threads within a node must be scalable, and the number of clusters supported by the TLS schema, which must also be scalable. I chose the snoopy protocol to support the TLS execution, which gives us scalability in terms of the number of clusters in our system. And the snoopy protocol also enforces the coherence of data cache across multiple nodes. And our speculative execution hardware provides the scalability of TLS execution within a node with only a small amount of hardware.

The following terms will be used in this chapter:

Epoch: Meaning an execution thread or a task.

Epoch number: A number is used to define the logical or program order of the epochs. This is a partial order. The epochs are committed in the order of their epoch numbers.

The meanings of these terms are same as [31] [68].

4.3. Cache Line States in Our Design

To support the TLS execution, the TLS schema must be able to detect the data dependence violations at run time. Using an MSI protocol, these violations can be easily detected.

I extend the standard invalidate-based MSI protocol. In order to support our TLS execution, two new states are included: the SpR.Sh and the SpR.Ex, in addition to the three states of MSI protocol (Excusive (E), Shared (S), and Invalid (I)). The SpR.Sh state means the cache line has been read by a speculative thread and more than one copy of this cache line exists in the system. The SpR.Ex state means that this is the only copy of cache line in the system and it has been read by a speculative thread. The data modified by a thread is contained in the register set assigned to the thread and the results are not committed to memory (or other thread frames) until the thread is allowed to complete its "post-store" portion of execution. The speculative thread are not allowed to write the data into cache, which will reduce the overhead of recovering from a data dependency violation. Table 4.3 includes all the cache line states in the TLS schema.

State	Description
M	Modified and exclusive ownership
S	Shared
I	Invalid
SpR.Sh	Speculative Read, Shared copy
SpR.Ex	Speculative Read, Exclusive copy

TABLE 4.1. Cache Line States of TLS Schema

In order to differentiate these states, it only needs one additional bit (more than the 2 bits needed to implement snoopy coherence protocol) with each cache line to identify these states. The three bits are: Invalid bit, Dirty (Modified) bit, and SpR (speculative read) bit. The Invalid bit and Dirty bit are the same as regular cache. SpR bit defines the speculative access of a cache line. Table 4.3 shows the encoding of the states.

The cache line is in the Invalidate state, if the Valid bit is 0. The cache line is in the Exclusive (Modified) state, if the Valid bit and the Dirty bit are 1 and the SpRead bit is 0. The cache line is in the Share state, if only the Valid bit is 1. The cache line is in the

	SpRead	Valid	Dirty(Exclusive)
I	X	0	X
E/M	0	1	1
S	0	1	0
SpR.Ex	1	1	1
SpR.Sh	1	1	0

TABLE 4.2. Encoding of Cache Line States

SpR.Ex state, if all the status bits are 1. And the cache line is in the SpR.Sh state, if the Valid bit and the SpRead bit are 1 and the Dirty bit is 0.

4.4. Continuation in Speculative SDF Architecture

In SDF architecture, a thread can be uniquely identified by a continuation, which is defined by $< FP, IP, RS, SC >$ (see Chapter 3). In order to support speculative execution of threads, I add three more elements into the continuation definition. A continuation in the SDF architecture which supports TLS is defined by: $< FP, IP, RS, SC, EPN, RIP, ABI >$. The first four elements are the same as in the original continuation definition in SDF. The added elements are: the epoch number (EPN), the re-try instruction pointer (RIP), and an address-buffer ID (ABI). For any TLS schema, an execution order of threads must be defined based on the program order. The epoch numbers (EPN) are used for this purpose. Speculative threads must commit in the order of their epoch numbers. RIP defines the instruction at which a failed speculative thread must start its retry. The other architectures that support speculation only allow a speculative thread to fail or succeed [31] [68]. But for SDF architecture with TLS support, if the speculation failed, instead of re-executing the thread from the beginning, I would start from a point where the first speculative read access is performed. Since the thread execution goes through pre-load, execute and post-store; any and all memory accesses take place in pre-load portion of the code, and a failed speculative thread will restart in the pre-load portion at the point a speculative access is made. RBI defines this point of restart. ABI defines the buffer ID that is used to store the addresses of speculatively-read data. I will explain the details of the ABI design in section 4.6. For the non-speculative thread, the three new fields will all be set to zero as

$< FP, IP, RS, SC, --, --, -- >$ so the hardware can easily determine whether a thread is speculative or not by testing the EPN field in a continuation.

4.5. Thread Schedule Unit in Speculative Architecture

The thread in SDF architecture moves between SPs and EPs and the thread schedule unit is responsible for scheduling waiting threads on SPs and EPs. In conventional SDF architecture, without speculative threads, there are two queues that pertain to threads awaiting to be scheduled on SPs: preload queue (enabled threads waiting to be scheduled on SPs for preload) and post-store queue (threads that have completed their execution and waiting to store results).

A separate queue are added for speculative threads to control the order of their commits. Figure 4.2 shows the overall design of the new architecture (compare with Figure 3.6). This queue is ordered by the EPN (epoch numbers) of the thread continuations.

For the controller (thread schedule unit) to distinguish between speculative and non-speculative threads, it only needs to test the epoch field of the continuation to see if it is equal to zero (as stated previously, a non-speculative thread's EPN is set to zero and any continuation that has a non-zero epoch number is a speculative thread). All the speculative threads are committed in order.

The commit controller maintains the epoch number of the next thread that can commit based on the program order. It will test the epoch number of a continuation that is ready for commit. If EPN of a thread matches this number and no data access violations are found in the address buffer associated with the thread, the commit controller will schedule the thread for commit (i.e. schedule the thread on SP for post-store). If there is a violation, the commit controller sets the IP of that continuation to RIP and places it back in the preload queue for re-execution. At this time, the thread becomes non-speculative.. The epoch number of the next thread that can commit is updated. It needs to note that the commit is a sequential operation according the order of the epoch number, which can become another bottleneck of future performance.

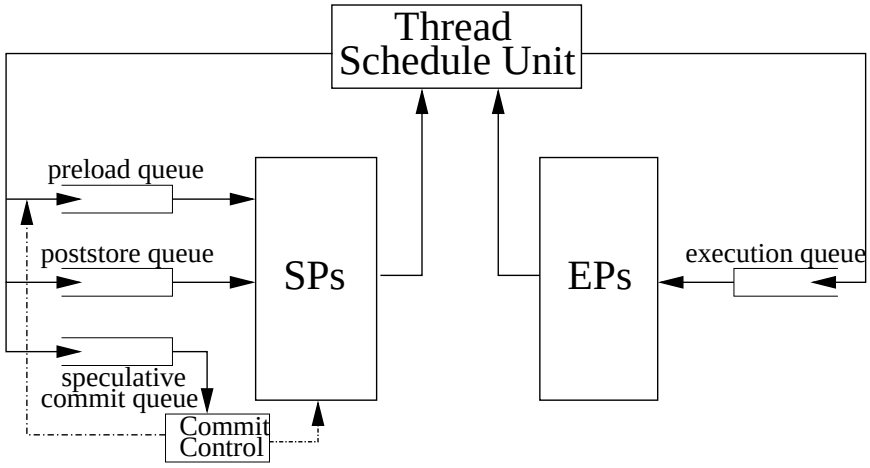


FIGURE 4.2. Overall TLS SDF Design

4.6. ABI Design

A few small, fully-associative set of buffers are used to record the addresses of data that are speculatively accessed by speculative threads. Data addresses are used as indices into these buffers. The small fully associative buffers can be implemented using an associative cache where the number of sets represents the maximum number of speculative threads and the associativity represents the maximum number of speculative data items that can be read by a thread. For example, a 64 set 4-way associative cache can support 64 speculative threads with four speculative address entries per thread. The address buffer ID (ABI) is assigned when a new continuation for a speculative thread is created. When a speculative read request is issued by a thread, the address of the data being read is stored in the address buffer assigned to the thread and the entry is set to valid. When a speculatively read data is subsequently written by a non-speculative thread, the corresponding entries in the address buffers are invalidated, preventing speculative threads from committing. The block diagram of address buffer for a 4-SP node is shown in figure 4.3. This design allows for invalidation of speculatively-read data in all threads simultaneously. It also allows different threads to add different addresses into their buffers. When and “invalidate” request comes from the bus or a ”write” request comes from inside the node, the data cache controller will change the cache line states, and the speculative controller will search the address buffer to invalidate appropriate entries.

Threads in SDF architecture are fine-grained and thus the number of data items speculatively read will be small. By limiting the number of data items speculatively read, the probability that a speculative thread successfully completes can be improved. For example, if p is the probability that speculatively read data will be invalidated, then the probability that a thread with n speculatively read data items will successfully complete is given by $(1-p)^n$. With four to eight speculative reads per thread and 16 speculative threads, it only needs 64 to 128 entries in the address buffers. Because the threads are non-blocking, they are allowed to complete the execution phrase even if some of the speculatively read data is invalidated. This eliminates complex mechanisms to interrupt threads, but may cause wasted execution of additional instructions for speculative threads.

The number of entries in the ABI determines the number of speculative threads that can be supported in an SDF node.

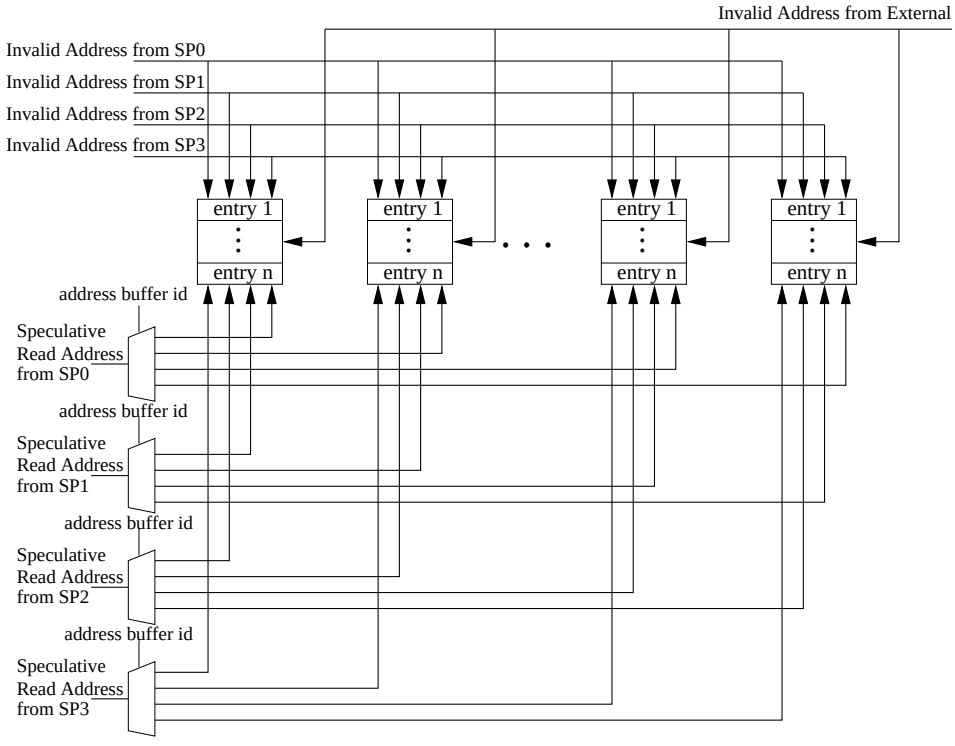


FIGURE 4.3. Address Buffer Block Diagram

4.7. State Transitions

In this section, I will introduce the actions and state transitions needed to implement the extensions to Cache Coherence in detail.

4.7.1. Cache Controller Action

In an invalidate-based cache coherence protocol, the cache controller will serve requests either from the processors or from the external memory.

The processor-initiated requests can be a combination of the reads or writes, hits or misses, and speculative or non-speculative accesses. This gives us eight total possible combinations. Because I do not allow the speculative thread to write, the number of the actions from the processors is reduced to six. The following table shows the action initiated by the processors.

Actions Initiated by Processors	Description
Read Hit (non-speculative)	Non-speculative thread read hit
SP Read Hit (speculative)	Speculative thread read hit
Read Miss (non-speculative)	Non-speculative thread read miss
SP Read Miss (speculative)	Speculative thread read miss
Write Hit (non-speculative)	Non-speculative thread write hit
Write Miss (non-speculative)	Non-speculative thread write miss

TABLE 4.3. Encoding of Cache Line States

The corresponding messages generated by the cache controller are the same as in the regular MSI protocol: Read Miss and Write Miss.

4.7.2. State Transitions According to the Processor Requests

In this section, I'm going to introduce the actions and state transitions. I will start with the each of the five states and examine the possible actions.

4.7.2.1. Invalid State

The only possible actions for this state are Read Miss, SP Read Miss, and Write Miss. For Read Miss, the cache controller will put a read miss message on the bus and wait for the data to come and then set the cache line state to Shared. For SP Read Miss, the cache controller will place a read miss on the bus. When the data comes back it will set the cache line to SP Shared state and add this address to the appropriate ABI associated with the thread. For Write Miss, the cache controller will send out a write miss message and acquire

exclusive ownership of this cache line and invalidate this address in local ABI. When the data comes back, it will update this cache line and set it to Exclusive state.

4.7.2.2. Exclusive State

In exclusive state, Read Hit and Write Hit will not generate any transitions. SP Read Hit will change the state to SP.Exclusive, and the address will be recorded in the ABI.

When Read Miss happens, the cache line will be written back and replaced by the new data. And then the state of this cache line will be set to Shared. When SP Read Miss happens, the cache line will be written back and replaced by the new data, and the address will be recorded into the local ABI and the state will be set to SP Shared. When Write Miss happens, the data will be written back, and a write miss message will be put on the bus and the address in the local ABI will be invalidated. Once the data comes back, the cache controller will update the data and set it to Exclusive state.

4.7.2.3. Shared State

Read Hit will not generate any actions. Read Miss will generate a read miss message; once new data comes back the cache line state will still be in Shared state. Sp Read Miss will generate a read miss message and the new address will be added to the ABI. Once the data comes back, the cache line state will be set to SP.Share (and the state is changed in all caches to SpShared). Write Hit will generate a write miss message to acquire exclusive ownership of the cache line and invalidate the address in local ABI; once the data comes back, the cache line will be replaced and updated, and the state will be set to Exclusive.

4.7.2.4. Sp.Exclusive State

Read Hit will generate no actions. Sp Read Hit will add the address to the ABI. Read Miss will cause the cache line to be written back and generate a Read Miss message; once the data comes back, the state will be set to Shared state. Sp Read Miss will perform an action similar to the Read Miss, but the difference is when the data comes back, the address will be recorded in the ABI and the cache line state will be set to Sp.Shared state. Write Hit will cause the cache controller to invalidate the local ABI and the state will change to Exclusive state. Write Miss will generate a write miss message on the bus and write this

cache line back. Once the new cache line comes back it will update the cache line and set it to the Exclusive state.

4.7.2.5. Sp.Shared State

Read Hit will generate no action. Sp.Read miss will add the address to the local ABI and there will be no state transition. Read Miss will cause a Read Miss message to be generated, and once the new data comes back, it will be put in the Shared state or Sp.Shared state if the address is in local ABI. Sp.Read Miss will also generate a Read Miss message. Once the data comes back, the address will be recorded in the local ABI and the states will still be Sp.shared. Write Miss will generate a Write Miss message to acquire exclusive ownership of the cache line and the address in the local ABI will be invalidated. Once the data comes, it will update the cache line and set it to Exclusive state.

Figure 4.4 shows the state transition diagram for the actions generated by the SPs. Table 4.4 and Table 4.5 summarize the state transitions.

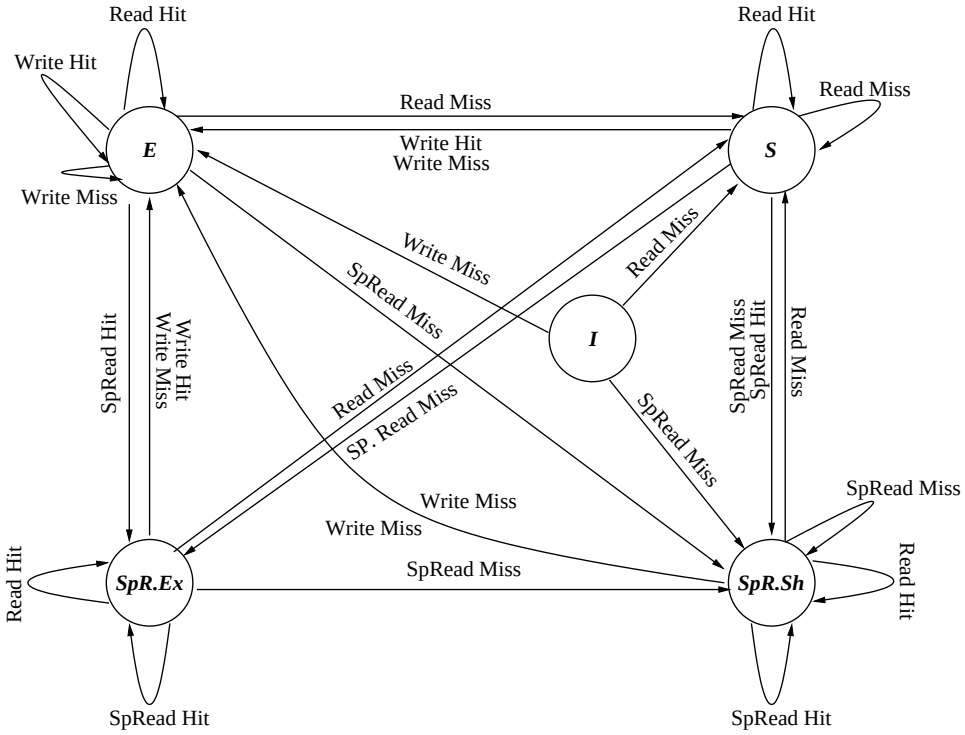


FIGURE 4.4. State Transitions for Action Initiated by SP

Cache Block State	Messages	Next Cache Block State	Actions
Invalid	Read Miss	Shared	Read Data from cache
	SpRead Miss	Sp. Shared	1. Read Data from cache 2. Update the ABI
	Write Miss	Exclusive	1. Send Write Miss message to acquire Exclusive ownership 2. Read the cache line 3. Update the cache line
Exclusive	Read Hit	Exclusive	1. Read Data from cache 2. Update entry to Epoch-Address buffer 3. Update entry in Address-Epoch buffer
	Read Miss	Shared	1. Write back the cache line 2. Send Read Miss message 3. Read in the cache line
	Write Miss	Exclusive	1. Write back the cache line 2. Send Write Miss message to acquire Exclusive ownership 3. Read in the cache line 4. Update the cache line
	Write Hit	Exclusive	No action
	Sp.Read Hit	Sp.Exclusive	Update the ABI
	Sp.Read Miss	Sp.Shared	1. Invalidate local ABI 2. Write the cache line back 3. Send Read Miss message 4. Read in the cache line 5. Update the ABI
Shared	Read Hit	Shared	No action
	Read Miss	Shared	1. Send Read Miss message 2. Read in the cache line
	Write Hit	Exclusive	1. Send Write Miss message to acquire Exclusive ownership 2. Update the cache line
	Write Miss	Exclusive	1. Send Write Miss message to acquire Exclusive ownership 2. Read the cache line 3. Update the cache line
	Sp.Read Hit	Sp.Shared	Update the ABI
	Sp.Read Miss	Sp.Shared	1. Send the Read Miss message 2. Read in the cache line 3. Update the ABI

TABLE 4.4. State Transitions Table for Action Initiated By SP (Part A)

4.7.3. State Transitions According to the Bus Requests

In this section, I will introduce the actions generated by the bus (caused by caches access in other nodes). There are only Read Miss and Write Miss messages on the bus. If local

Cache Block State	Messages	Next Cache Block State	Actions
Sp.Shared	Read Hit	Sp.Shared	No action
	Read Miss	Shared	Update ABI (Invalid)
	Write Hit	Exclusive	1. Send Write Miss message to acquire Exclusive ownership 2. Update the cache 3. Update ABI (Invalid)
	Write Miss	Exclusive	1. Update ABI (Invalid) 2. Send Write Miss message to acquire Exclusive ownership 3. Read in cache line 4. Update cache
	Sp.Read Hit	Sp.Shared	Update ABI
	Sp.Read Miss	Sp.Shared	1. Update ABI (invalid) 2. Send Read Miss message 3. Read in cache 4. Update ABI
Sp.Exclusive	Read Hit	Sp.Exclusive	No action
	Read Miss	Shared	1. Update ABI (invalid) 2. Write back the cache line 3. Send Read Miss message 4. Read the cache line
	Write Hit	Exclusive	Update ABI (invalid)
	Write Miss	Exclusive	1. Update ABI (Invalid) 2. Write back the cache line 3. Send Write Miss message to acquire Exclusive ownership 4. Read in cache line 5. Update cache line
cline2-4	Sp.Read Hit	Sp.Exclusive	Update ABI
cline2-4	Sp.Read Miss	Sp.Shared	1. Update ABI (Invalid) 2. Write back the cache line 3. Send Read Miss message 4. Read in cache line

TABLE 4.5. State Transitions Table for Action Initiated By SP (Part B)

cache does not have that cache line, no action will be performed for Read Miss and invalid local ABI will be performed for Write Miss. If the cache line has the data, I will examine the state of each of the four states.

4.7.3.1. Shared State

There will be no action to Read Miss. To Write Miss, the cache line will be invalidated.

4.7.3.2. Sp.Shared State

There will be no action to Read Miss. To Write Miss, the cache line and corresponding ABI entries are invalidated.

4.7.3.3. Exclusive State

For Read Miss, the cache line is written back and the cache line state is changed to Shared state. For Write Miss, cache line is written back, and the local cache line is set to Invalid state.

4.7.3.4. Sp.Exclusive State

For Read Miss, the cache line is written back and the cache line state is changed to Shared state. For Write Miss, the cache line is written back; the local cache line is changed to Invalid state and entries in the local ABI are invalidated.

Figure 4.5 shows the state transition diagram for actions generated by the bus.

The key idea is that every speculative read will change the cache line state to speculative and also allocate an entry to the corresponding ABI buffer and every (non-speculative) write will invalidate the entries in the ABI buffer and the states of cache lines are modified in accordance to invalidate-based cache coherency protocol.

Table 4.6 summarizes the state transitions.

Cache Block State	Messages	Next Cache Block State	Actions
Invalid	Read Miss Write Miss	Invalid	No Action
Exclusive	Read Miss	Shared	No Action
	Write Miss	Invalid	Write back the cache line
Shared	Read Miss	Shared	No action
	Write Miss	Invalid	No action
Sp.Shared	Read Miss	Sp.Shared	No action
	Write Miss	Invalid	Update ABI (Inavlid)
Sp.Exclusive	Read Miss	Sp.Shared	No action
	Write Miss	Invalid	1. Update ABI (Inavlid) 2. Write back the cache line

TABLE 4.6. State Transitions Table for Action Initiated By Memory

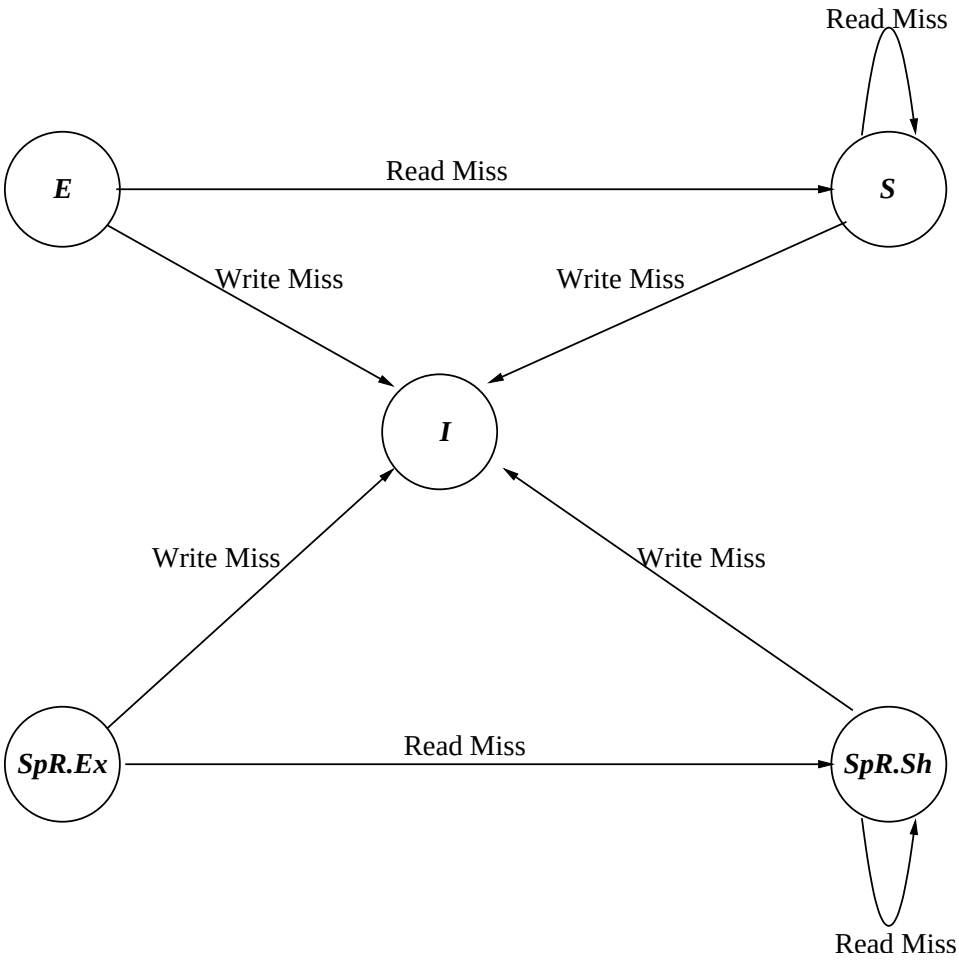


FIGURE 4.5. State Transitions Initiated by Memory

4.8. ISA Support for SDF architecture

I added five new instructions to the SDF instruction set for thread-level speculation support.

The first instruction, SFALLOC, is for allocating a frame to a speculative thread. This instruction will request the system to assign an epoch number and an ABI for the new speculative thread continuation.

The SFORKSP and SFORKEP instructions will be used by the system to indicate the moving of a speculative thread between SP and EP.

The SREAD instruction is for speculatively reading data, which will cause an entry to be added into the address buffer associated with that continuation. It should be noted that not all reads of a speculative thread are speculative reads. A compiler can resolve most data dependencies and use speculative reads only when static analyses cannot determine memory

ordering. It should also be noted that when a speculative thread is invalidated, the retry needs only to re-read speculatively-read data.

The SCFORKSP instruction is for committing a speculative thread. This instruction will place the speculative thread continuation into the speculative thread commit queue.

4.9. Compiler Support for SDF Architecture

For the simplest case, a compiler can generate speculative tasks exclusively out of loop iterations [26] [80]. The reason is that loops are often the best source of parallelism and the algorithm used to analyze the cross loop iteration data dependencies is very mature. For a slightly complicated compiler system, a dependence analysis phase identifies the most likely data dependences in the code and partitions the code into tasks to minimize inter-task dependences. If there are ambiguous data dependencies between tasks, the compiler can also utilize the speculative thread execution model to maximize the parallelisms [15] [42] [76]. The most aggressive compiler system can utilize the speculative thread execution at the procedure level. When a function call is encountered, the compiler can generate two threads: one non-speculative thread executes the function call and the speculative thread continues the current execution path. By doing this, the granularity of the speculative thread will be increased and the overhead of speculative thread generation will be reduced [55] [62].

Due to the fine-grained thread characteristic of SDF system, a compiler, which can analyze the data dependencies at loop level, will successfully generate the speculative thread for SDF system. Most of today's compilers have the module to analyze data dependencies at the loop level.

CHAPTER 5

SDF WITH TLS EXPERIMENTS AND RESULTS

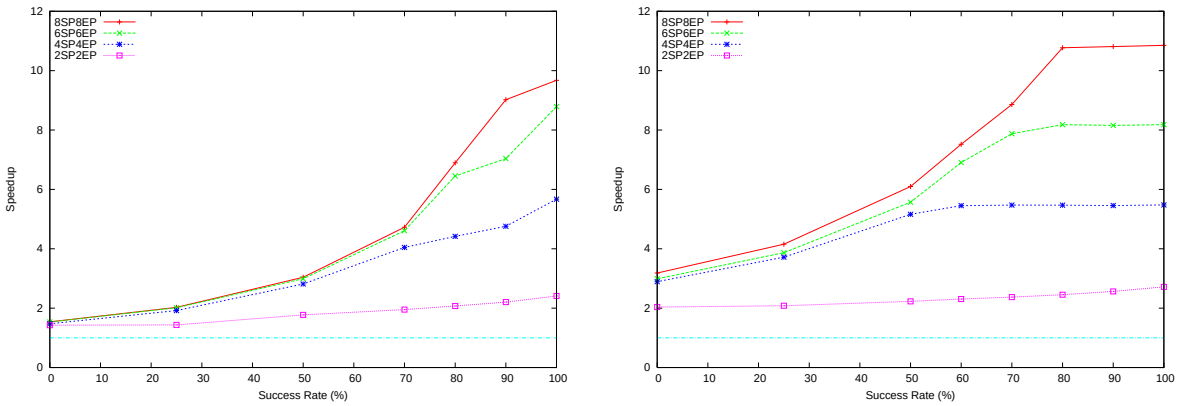
I implemented the speculative thread execution schema within the SDF simulator framework. The existing SDF simulator performs a cycle-accurate functional simulation of SDF instructions.

5.1. Synthetic Benchmark Results

In order to model the performance of the speculative thread execution schema, I first created a synthetic benchmark that executes a loop containing a variable number of instructions. I controlled the amount of time a thread spends at SPs and EPs by controlling the number of LOADS and STORES (workload on SP) and computational instructions (workload on EP). The base performance are collected using a non-speculative execution of these threads. Then, I used the TLS to parallelize these benchmarks. I tested this group of benchmarks both in terms of scalability - number of SPs and EPs in the system, and the success rate of the speculative threads - and correct speculation.

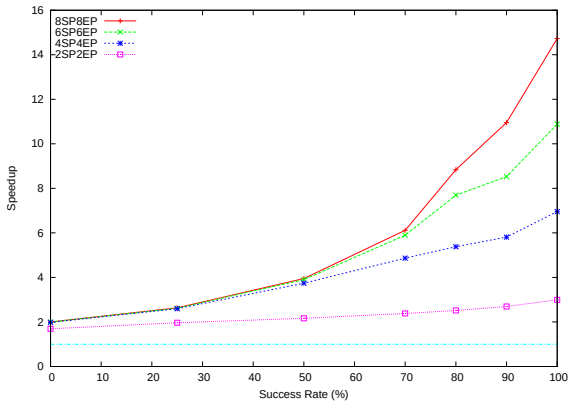
Figure 5.1(a) shows the performance of a synthetic benchmark that spends 33% of the time at SPs and 67% of the time at EPs, when executed without speculation. Figure 5.1(b) shows the performance for a program with 67% SP workload, 33% EP workload, while Figure 5.1(c) shows the data for programs with 50% SP and 50% EP workloads (if executed non-speculatively). All programs are tested using different speculation success rates. I will show data with different numbers of functional units: 8SPs-8EPs, 6SPs-6EPs, 4SPs-4EPs, and 2SPs-2EPs.

Since the SDF performs well when the SPs and EPs have a balanced load (and achieve optimal overlap of threads executing at EPs and SPs), we would expect the best performance for the cases shown in Figure 5.1(c) and when the success of speculation is very high (closer to 100%). However, even if I started with a balanced load, as the speculation success drops (and is closer to zero), the load on the EPs increases because failed threads will have to re-execute



(a) Benchmark Spend 33% of the Time on SP, 67% on EP

(b) Benchmarks Spend 67% of the Time on SP, 33% on EP



(c) Benchmarks Spend 50% of the Time on SP, 50% on EP

FIGURE 5.1. The Performance and Scalability of Different Load and Success Rates of TLS Schema

their computations. As stated previously, a failed thread only needs to re-read the data items that were read speculatively, and data from a thread are post-stored only when the thread speculation is validated. Thus a failed speculation will disproportionately add to the EP workload. For the case shown in Figure 5.1(b) with a smaller EP workload, I obtain higher speed-ups (compared with Figures 5.1(a) or 5.1(c)) even at lower success rates of speculation, since EPs are not heavily utilized in this workload. For the 33% – 66% SP-EP workload in Figure 1(a), even a very high success rate will not lead to high performance gains on SDF because EPs are already overloaded, and the mis-speculative thread will further increase the load of EPs. From Figure 5.1(c), we can also see that when the speculation success is below 50%, there are insignificant differences among the performance gains resulting from

the different number of functional units (number of EPs and SPs), except for the 2SP-2EP setup. This is because, at lower success rates, there are insignificant opportunities for realizable parallelism.

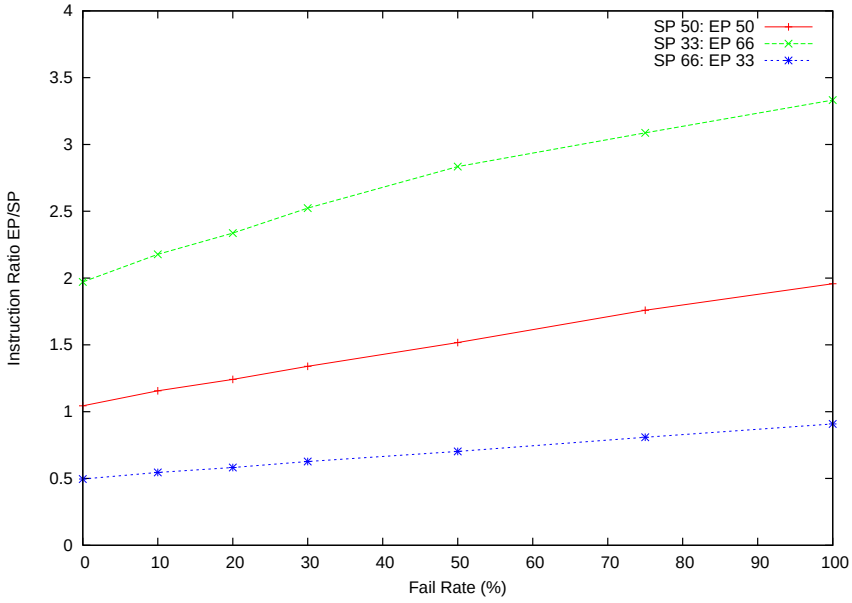


FIGURE 5.2. Ratio of Instruction Executed EP/SP

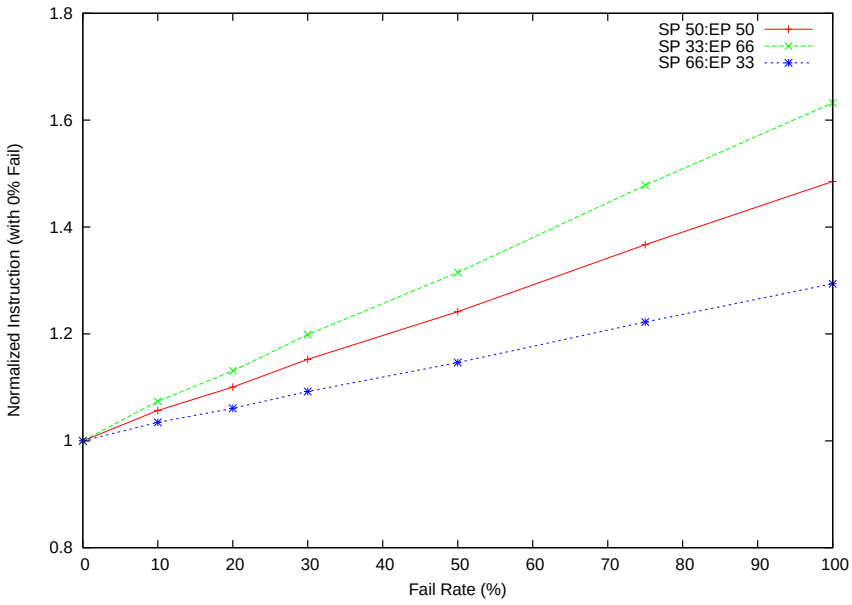


FIGURE 5.3. Normalized Instruction Ratio to 0% Fail Rate

Figure 5.2 below shows the ratio of actual instructions executed by EPs over the instructions executed by SPs (y-axis) as the speculation failure rate increases (x-axis). With an

increasing failure rate, the EPs will be more heavily loaded (higher ratios on the y-axis), and the performance will drop as the success rate drops. The balance of the workload between EP and SP will affect scalability. Even with higher speculation success rates, the workload SP33:EP66 (with higher EP load) has poorer performance than the other workloads. Figure 5.3 shows the increase in the total workload (combined EP+SP instructions executed), normalized to the case when the speculation success is 100% (y-axis). Since most of the re-try work is done by the EP, a smaller EP/SP work ratio will result in a smaller increase in the overall work, with increasing mis-speculation rates. When the speculation success approaches zero, (indicating strict sequential ordering of threads,) the overhead due to retries can actually cause a performance drop in a multi-threaded system. However, the SP66:EP33 workload, which has the smallest increases in the overall workload, has the best performance when the speculation success rate is below 50%.

In thes experiments, I show that even with very small success rates, the thread-level speculation can lead to some performance gains. This performance gain is due to the fact that I start later iterations of a loop earlier. Each iteration has a non-speculative part and a speculative part. The non-speculative parts can execute in parallel. Hurson et al. [6] describes a compiler approach to split loops into parallel and sequential loops - the parallel loop can be executed fully in parallel. This is effectively achieved by the TLS scheme where speculative threads complete the parallel portions of their computations, and may have to re-do the sequential portions of their computations on a mis-speculation.

From these experiments, we can draw the following conclusions. Speculative thread execution can lead to performance gains over a wide range of speculation success probabilities. At least a 2-fold performance gain can be expected, when the success of speculation is greater than 50%. If the success rate drops below 50%, one should turn off speculative execution to avoid excessive retries that can overload the EPs. When the EP workload is less than the SP workload, the TLS schema can tolerate higher rates of mis-speculation. Finally, when the success rates are below 50%, the performance does not scale well with added SPs and EPs (8SPs-8EPs, 6SPs-6EPs, and 4SPs-4EPs all show similar performance). This suggests that the success of speculation can be used to decide on the number of SPs and EPs needed to achieve optimal performance.

5.2. Real Benchmarks

To further test my design, I selected a set of real benchmarks. I hand-coded these benchmarks using SDF assembly language. This group of benchmarks includes: Livermore loops 2 and 3, two major functions compress() and decompress() from 129.compress benchmark, and four loops chosen from 132.ijpeg benchmark. Table 5.1 shows the detailed description of the benchmarks. I coded these benchmarks in two forms: one without speculation, where all the threads are executed sequentially, and the other with speculation. In the speculative execution, earlier iterations (or threads with lower epoch numbers) generated speculative threads for later iterations (or threads with higher epoch numbers).

Suite	Application	Selected Loops
Livermore Loops		Loop2 Loop3
SPEC 95	129.Compress95	Compress.c:480 while loop Compress.c:706 while loop
	132.ijpeg	Jccolor.c:138 for loop Jcdectmgr.c:214 for loop Jidctint.c:171 for loop Jidctint.c:276 for loop

TABLE 5.1. Selected Benchmarks

I evaluated performance gains using different numbers of SPs and EPs and the results are shown in figure 5.4. The speculative execution does achieve higher speedups - between 30% and 158% for the 2SP-2EP configuration and between 60% and 200% speedup for the 4SP4EP configuration. To compare my results with those of [68], I used the parallel coverage parameter defined in [68]. Using the 4SP4EP configuration to compare with their four tightly-coupled, single-threaded superscalar pipeline processors, for compress95 I achieved a speedup of 1.94 compared to 1.27 achieved by [68], and a speedup of 2.98 for ijpeg compared to 1.94 achieved by [68].

Another finding from figure 5.4 is that the performance does not scale well after the 4SP4EP configuration. This is because of the way I generated threads - I generated a very limited number of speculative threads, since each iteration only generates one new

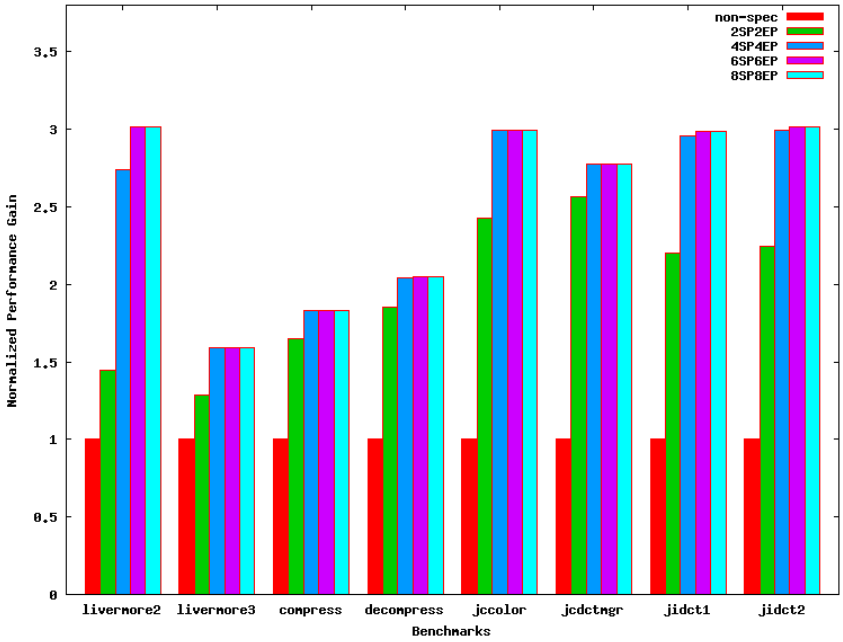


FIGURE 5.4. Performance Gains Normalized to Non-Speculative Implementation

speculative thread. However, with an optimizing compiler, it will be possible to generate as many speculative threads as needed to fully utilize available processing and functional units.

These experiments are repeated with the same benchmarks, but using a control thread that spawned multiple speculative threads at a time. For the Livermore loops, the control thread spawned 10 iterations at a time, and for the compress95 and the jpeg, the control thread spawned 8 iterations at a time. The results are shown in figure 5.5. In most cases, this approach does show better scalability with added functional units. Livermore loop 3 and compress are the exceptions. For these applications, the mis-speculation is very high and since on mis-speculation threads become non-speculative, (executing sequentially,) the available concurrency is reduced. It should be noted, however, that this approach does lead to higher speedups than those reported in [68].

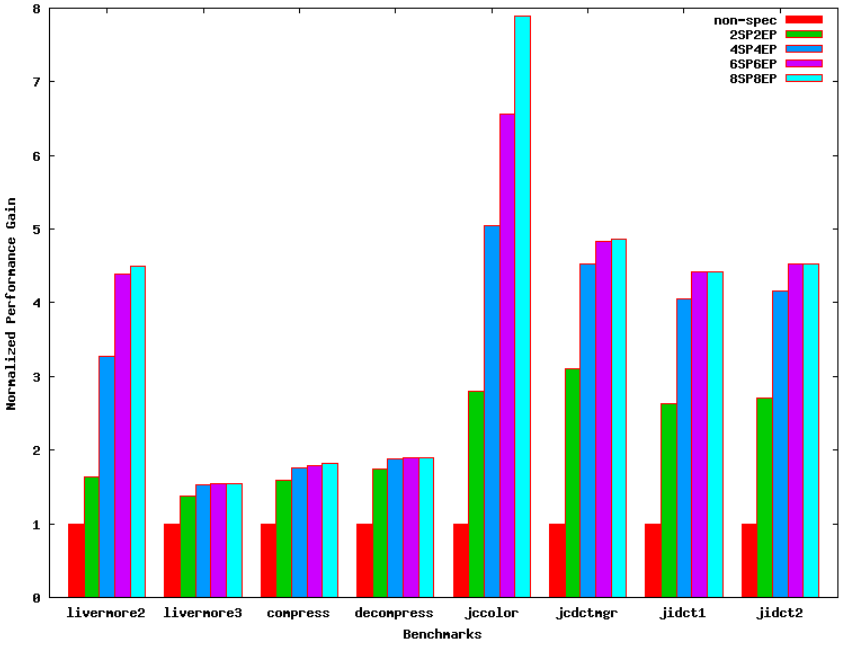


FIGURE 5.5. Performance Gains Normalized to Non-Speculative Implementation

CHAPTER 6

PERFORMANCE OF HARDWARE MEMORY MANAGEMENT

To achieve better performance with SDF architecture, a separated hardware unit are used to control the I-Structure management. In conventional computer systems, dynamic memory management functions are performed in software. The experience with SDF architecture suggests that conventional architecture can also benefit from using separate hardware memory manager or hardware-assisted memory manager to achieve high performance. In this chapter and the following two chapters I will demonstrate the usefulness of the hardware memory manager. This chapter will provide the background on memory management and a simple analysis of why hardware allocator improves performance; chapter 7 will provide the performance impact of different memory management algorithms when implemented in hardware; chapter 8 will show a simple design of a hardware-assisted memory manager that improves performance by 23%.

6.1. Review of Dynamic Memory Management

Dynamic memory management is an important problem that has been studied by researchers for several decades [78]. With the popularity of object-oriented languages such as C++ and Java, dynamic memory allocation and garbage collection can become significant barriers to application performance. My research goal is to explore the use of separate hardware for dynamic memory management. Several related research directions have influenced this work. I will briefly review the most relevant research in this section.

Dynamic memory management is traditionally implemented as software within a system's run-time environment. Modern programming languages and applications are driving the need for more efficient implementations of memory management functions, both in terms of memory usage and execution performance. Several researchers have proposed and implemented custom allocators and garbage collectors to improve performance of applications requiring dynamic memory management. Such allocators attempt to optimize allocations based on

an application's memory usage patterns. Berger, et al., describe a comprehensive evaluation of different memory allocators for a wide range of benchmarks including SPECint2000 and memory intensive applications [9]. Their study found that the total execution time spent on memory management functions can be significant. This finding supports the efforts to decouple memory management functions from the primary execution engine. Previously our group [63] have studied cache behavior and pollution of the cache due to the bookkeeping meta-data used by the allocation functions. This research suggests that if separate hardware logic is used to perform memory management functions, cache performance of the application can be improved.

Furthermore, while separate memory management hardware is important for many architectural paradigms, it is critical for Simultaneous Multithreading (SMT) architectures [74]. SMT places heavy demands on instruction and data caches [39]. By separating memory management functions from the execution pipeline, we can at least eliminate cache contention produced by memory management functions. This, in turn, improves the scalability of SMT architectures.

In this part of the work, I will focus on why separate hardware can improve performance instead of how to implement a separate hardware memory manager. This hardware unit can be integrated along with the CPU, incorporated within a memory controller, or embedded with IRAM [28].

6.2. Experiments

To evaluate the potential for decoupling memory management, the experiments are constructed to reflect the exact conditions as closely as possible. I identified and controlled experimental parameters such as machine model(s), appropriate benchmarks, and statistical attributes of interest. In this section, I will describe the methodology and the selection of benchmarks.

6.2.1. Simulation Methodology

For the purpose of studying the performance implications of decoupling memory management, I extended the SimpleScalar/PISA Tool Set, version 3 [13]. I assumed the existence

of hardware for memory management in the form of a separate hardware unit. The simulated memory management hardware behaves in the same fashion as Lea's allocator [53] used in a LINUX system. I further assumed that the hardware memory management unit is not pipelined. The later allocation or de-allocation request must wait for the previous requests to finish. In a real implementation though, one can use pipelined hardware for memory management to process "bursts" of allocation requests, particularly for applications that allocate several objects together. I added two instructions, "ALLOC" for allocation and "DEALLOC" for de-allocation, to the instruction set of SimpleScalar. The two new instructions are treated in the same manner as other PISA instructions and processed by scheduling them on the separate hardware allocator, viewed as a functional unit (similar to an Integer or Floating point unit). However, when allocation and de-allocation instructions are encountered, the reservation stations are frozen until the memory management completes and returns the results. This results in CPU stalls, particularly when using a slow hardware allocator. In an actual implementation, this restriction can be eliminated with proper hardware/software optimizations. Table 6.1 summarizes the simulation parameters. To explore the feasibility of this decoupling architecture, I used a wide range of allocation latencies (i.e., time to complete an allocation request and return the address of the allocated object), from 1-cycle to the number of cycles that match the software allocator in various experiments.

6.2.2. Benchmarks

Table 6.2 shows the benchmark programs used in the experiments. I selected the benchmarks for my experiments from SPECInt2000, memory intensive benchmarks, and Olden benchmarks. Table 6.2 also shows the total number of instructions executed by the benchmarks.

The selected benchmarks demonstrate different levels of memory management operations, as a percentage of total execution times (as shown in Table 6.3). These levels range from very high (parser, cfrac, treeadd), to average (espresso, voronoi), and to very low (vortex, gzip, bisort).

Table 6.3 lists the fraction of execution time spent on memory management functions. Looking at the fraction of time spent on memory management functions, one might assume

Pipelined CPU Parameters	
Issue Width	4
Functional Units	5 Int (4 ALU, 1 Mult/Div), 5 FP (4 ALU, 1 Mult/Div), 2 Memory, 1 Allocator, 1 Branch
Register Update Unit Size (RUU)	8
Load/Store Queue size (LSQ)	4
Integer ALU	1 cycle
Integer Multiply	4 cycles
Integer Divide	20 cycles
FP Multiply	4 cycles
FP Divide	12 cycles
Branch Prediction Scheme	Bimodel
Memory Parameters	
L1 Data Cache	4-Way set associative, 16 Kbytes
L1 Data Cache	4-Way set associative, 16 Kbytes
L1 Instruction Cache	Direct Mapped, 16 Kbytes
L2 Unified Cache	4-Way set associative, 256 Kbytes
Line Size	32 Bytes
L1 Hit Time	1 cycle
L1 Miss Penalty	6 cycles
Memory Latency/Delay	18/2 cycles
Allocation Time	100 or 1 cycles
De-allocation Time	1 cycle

TABLE 6.1. Simulation Parameters

that this limits the performance gains of decoupled architecture (i.e., the maximum performance gains using a hardware allocator are limited by the fraction of the time spent on memory allocation functions). However, several complex features of modern architectures impact performance. These factors include cache misses, pipeline stalls, out-of-order execution, and speculations. I will show that these factors may lead to performance gains greater than the fraction of cycles spent on memory management functions (see the last column of Table 6.3).

6.3. Experiment Results and Analysis

In this section, I will report the results of the experiments. I will discuss both the execution performance and cache behavior resulting from decoupling memory management functions.

6.3.1. Execution Performance Issues

6.3.1.1. 100-Cycle Decoupled System Performance

I assume that each malloc operation takes a fixed 100 cycles in this experiment. Table 6.4 shows the performance improvements achieved when a separate hardware unit is used for all memory management functions (malloc and free). The second column in the table shows the number of cycles needed for a conventional architecture and the third column shows the execution cycles needed by a decoupled system. The fourth column shows the percentage of speed-up achieved by the architecture. The fifth column reproduces the fraction of cycles (from Table 6.3) spent on memory management functions; it is named as Percentage of Cycles in Memory Management (CMM for short). The last two columns of the table depict the utilization of superscalar in terms of the instruction per cycles (IPC). In both cases, the IPC does not exceed 1.67. The instruction count of a decoupled system is smaller than the conventional architecture, since software implementation of the memory management functions is replaced by the hardware. Smaller IPCs in the decoupled system can be due to the non-pipelined implementation of the memory management hardware and the freezing of

Benchmark Family	Benchmark Name	Benchmark Description	Input	No. Inst. (Million)
SPEC	164.gzip	gnu zip data compression	test	4,540
	197.parser	English parser	test	1,617
	255.vortex	object oriented database	test	12,983
MEM	cfrac	factoring numbers	a 22 digit No.	96
	espresso	PLA optimizer	mpl4.espresso	73
OLDEN	bisort	sorting bitonic sequences	250K integers	607
	treeadd	summing values on a tree	1M nodes	95
	voronoi	computing voronoi diagram	20K points	166

TABLE 6.2. Description of Benchmarks

Benchmark Name	% Execution Time in Memory Management
164.gzip	0.04
197.parser	20.65
255.vortex	0.59
cfrac	18.75
espresso	11.63
bisort	2.08
treeadd	49.44
voronoi	8.75

TABLE 6.3. Percentage of Time Spent on Memory Management Functions

reservation stations on malloc requests. These restrictions limit the amount of Instruction Level Parallelism (ILP). This means that the number of eliminated cycles is less than the number of eliminated instructions.

Benchmark	CC (Cycle Count) - Million		Speedup	CMM	IPC (Inst. Per Cycle)	
Name	CONV	Decoupled	%	%	CONV	Decoupled
164.gzip	2,725	2,724	0.0309	0.04	1.67	1.67
197.parser	1,322	1,280	3.19	20.65	1.57	1.26
255.vortex	12,771	12,602	1.34	0.59	1.00	1.03
cfrac	107	99	7.83	18.75	1.16	0.96
espresso	46	45	1.44	11.63	1.18	1.46
bisort	474	426	10.03	2.08	1.31	1.42
treeadd	134	165	-23.19	49.44	1.59	0.58
voronoi	123	123	-0.01	8.75	1.38	1.23

TABLE 6.4. Execution Performance of Separate Hardware for Memory Management

Before discussing the range of speed-ups achieved using a slow allocator (fourth column of Table 6.4-100-cycle Decoupled), I should re-emphasize that 100 cycles for a hardware implementation of memory management functions implies a slow hardware. In contrast, Chang et al. describe hardware that requires, on average, 4.82 cycles (note that they did not mention whether this is CPU or memory cycle) [14]. With the slow implementation using

100 cycles, it is possible for the CPU to idle waiting for a memory allocation, reflected by lower IPC counts.

Two anomalies are noticed when examining the speedup achieved using 100-cycle hardware implementation (fourth column of Table 6.4). First, for some benchmark programs (vortex, bisort), even a 100-cycle hardware memory manager achieves higher performance than the fraction of cycles spent on memory management functions by a software implementation (comparing columns 4 and 5 of Table 6.4). In a moment, I will show that this is in part due to the CPU cache misses eliminated by moving memory management to dedicated hardware. The second anomaly is for voronoi and treeadd programs; the decoupled system shows performance degradation for these benchmarks.

I believe that this is due to two factors: (1) CPU stalls resulting from a slow allocator (compare the IPCs), and (2) the allocation behavior of the application. The software allocator (Lea's) takes advantage of the allocation behavior of these programs. These programs perform all of their allocations at the beginning of the execution and keep all the allocated memory throughout the execution. In addition, most allocated objects are small and belong to 8, 16 or 32 byte chunks. These sizes can be allocated very fast in Lea's allocator.

6.3.1.2. 1-Cycle Decoupled System Performance

Table6.5 shows the execution speed-up achieved assuming 1-cycle for all memory management functions. This data places an upper bound on performance improvement for decoupled memory management architecture. I will discuss some techniques for achieving faster allocators later in this paper. Such implementations would eliminate CPU stalls awaiting an allocation, since allocations take only one cycle.

Note that eliminating the CPU stalls using a 1-cycle hardware implementation produces a "super-linear" speedup for almost all the benchmarks (fourth column, compared with the fifth column of Table 6.5). The speedup for the 1-cycle decoupled system should be at least the same as the percentage of cycles spent in memory management (CMM) in conventional architecture. This can be viewed as linear-speedup. If the percentage of speedup is greater than the percentage of the CMM, the system has achieved a super-linear speedup. According to the data shown in Table 6.5, a 1-cycle decoupled system reveals super-linear speedup for

Benchmark	CC (Cycle Count) - Million		Speedup	CMM	IPC (Inst. Per Cycle)	
Name	CONV	Decoupled	%	%	CONV	Decoupled
164.gzip	2,725	2,724	0.03	0.04	1.67	1.67
197.parser	1,322	1,074	18.81	20.65	1.57	1.51
255.vortex	12,771	12,591	1.41	0.59	1.00	1.03
cfrac	107	78	27.65	18.75	1.16	1.23
espresso	46	39	14.85	11.63	1.18	1.67
bisort	474	413	12.76	2.08	1.31	1.47
treeadd	134	63	52.65	49.44	1.59	1.51
voronoi	123	111	10.37	8.75	1.38	1.37

TABLE 6.5. Limits on Performance Gain

almost all the applications except gzip and parser. I attribute the super-linear performance to the removal of conflict (cache) misses between the memory allocation functions and the applications. In section 5.2, I also investigated the first level cache performance of the selected benchmarks.

6.3.1.3. Lea-Cycle Decoupled System Performance

Table 6.6 shows the average number of cycles spent per malloc call when a software implementation of the Lea's allocator is used. Note that the second column of Table 6.6 shows the average number of CPU cycles per memory management function (not the percentage shown in the other tables thus far). In the experiments thus far I have used a fixed number of cycles (either 100 or 1) for each allocation. However, as shown in Table 6.6, allocators take different amounts of time for allocation, depending on the size of the object and the amount of search needed to locate a chunk of memory sufficient to satisfy the request. I repeated the experiments using the same average number cycles for a hardware allocator as that for the software implementations respectively (second column of Table 6.6). The performance gains of these experiments are shown in the third column of Table 6.6.

Benchmark Name	Average CMM	% of Speedup
164.gzip	790	0
197.parser	69	10.02
255.vortex	401	0.88
cfrac	93	8.8
espresso	87	4.08
bisort	90	10.95
treeadd	67	0
voronoi	79	2.22

TABLE 6.6. Average Number of Malloc Cycles Needed by Lea Allocator

Based on the data shown in Table 6.6, we can classify these benchmarks into three groups. The first group consists of benchmark with an average number of cycles per memory management request exceeding 100 cycles (gzip and vortex). For these types of benchmarks, the performance of Lea's allocator is poor since they allocate objects with very large size. Lea's allocator has to request memory from the system for each large object. The second group includes the majority of the benchmarks and requires less than 100 cycles per memory management request. They include parser, cfrac, espresso, bisort, and voronoi. For these applications, even when the number of cycles needed per memory allocation by the hardware allocator is set equal to those of a software allocator, the performance gained by the decoupled allocator is noticeable. The third group of applications includes treeadd and generates allocation requests in a burst (several allocation requests in sequence). For these applications, the current hardware allocator causes CPU stalls since memory management hardware is not pipelined, resulting in performance degradations.

6.3.1.4. Cache Performance Issues

Previously I stated that the "super-linear" speedup with separate 100-cycle hardware for memory management functions (at least for vortex and bisort) is due in part to the elimination of CPU cache misses. Now, I explore this in more detail. Table 6.7 and 6.8 show the data for L-1 instruction and data caches.

Benchmark Name	Conventional Architecture		Decoupled	
	No. of Ref. (Million)	No. of Misses (Thousand)	No. of Ref. (Million)	No. of Misses (Thousand)
164.gzip	5,145	70,412	5,144	70,356
197.parser	2,320	10,841	1,825	6,040
255.vortex	14,148	974,678	14,094	959,584
cfrac	140	7,048	107	4,122
espresso	86	1,286	77	779
bisort	697	1.1	700	1.08
treeadd	257	1.3	124	0.98
voronoi	187	1,023	174	1,214

TABLE 6.7. L-1 Instruction Cache Behavior

Benchmark Name	Conventional Architecture		Decoupled	
	No. of Ref. (Million)	No. of Misses (Thousand)	No. of Ref. (Million)	No. of Misses (Thousand)
164.gzip	1,504	37,616	1,504	37,577
197.parser	927	11,659	677	8,298
255.vortex	6,920	70,412	6,875	68,828
cfrac	50	10	37	9.9
espresso	23	94	20	74
bisort	161	2,193	156	2,193
treeadd	88	1,068	40	1,056
voronoi	58	1,054	53	928

TABLE 6.8. L-1 Data Cache Behavior

The reduction in instruction cache misses can be more easily understood since instructions comprising malloc and free are removed from the execution pipeline. The reduction in data references and misses (Table6.8) is because the allocation bookkeeping meta-data maintained

by the allocator is no longer brought into CPU cache. These results are similar in spirit to those of [9], but differ in actual values.

Using miss penalties from SimpleScalar, as well as the memory accesses eliminated (both from Instruction and Data caches), one can estimate the number of cycles eliminated from CPU execution. This should indicate the performance contribution due to improved CPU cache performance. For example, for vortex, the elimination of some memory accesses for instructions and data as well as the reduction in cache misses has contributed to 2% of the 2.81% improvement shown in Table 6.4; the remaining performance is mostly due to the elimination of instructions from the execution pipeline. Note that for vortex, since this application shows a CPI close to one cycle on average, computing the contribution of reduced cache misses to the overall performance gains is straightforward.

Similar computations can be used to find the performance gains due to improved cache hits for other benchmarks; however, such computations are more complex because an IPC that is not equal to one reflects out-of-order execution of instructions. This explains most of the differences between the percentage of time spent on dynamic memory management in a conventional architecture (fifth column of Table 6.5) and the potential performance improvement with a 1-cycle hardware allocator (fourth column of Table 6.5). Other factors such as out-of-order-execution and speculative execution change the instructions per cycle (IPC) counts for the architecture.

CHAPTER 7

ALGORITHM IMPACT OF HARDWARE MEMORY MANAGEMENT

Dynamic memory management is an important problem that has been studied by researchers for the past several decades. Modern programming languages and applications are driving the need for more efficient implementations of memory management functions, both in terms of memory usage and execution performance. Several researchers have proposed and implemented custom allocators and garbage collectors to improve performance of applications requiring dynamic memory management.

In general, a hardware implementation of any function should require fewer CPU cycles than its software counterpart. When memory management functions are implemented in software, the chosen allocator algorithm impacts application performance due to the following behavioral differences of the allocators:

a) The number of instructions required by allocators differ since different search and allocation methods are used (for example, first fit, best-fit, segregated lists, buddy lists, etc).

b) The number of instruction cache accesses and misses caused by allocator functions differs among allocators and choice of allocator may also cause different numbers of cache conflicts with applications' instruction accesses.

c) The number of additional data cache accesses and cache conflicts caused software allocators. The software allocator needs to access bookkeeping data, such as object headers. Bookkeeping data must be brought into CPU cache for allocator use, causing more data accesses and misses (application data + bookkeeping data), more cache conflicts among application data, and additional allocator bookkeeping data.

d) The localities of allocated objects. .Different allocator algorithms assign objects to different memory areas, leading to different localities of allocated objects.

As I will show in this dissertation, when using hardware allocators, the performance of applications is affected mostly by item (d) above, and the actual allocator method used has less significant impact on applications' performance, implying that one can select an

allocator based either on ease of implementation, or on differing localities of applications' objects.

In this chapter, I compare performance and cache behaviors of three general purpose allocators: Lea's allocator [53] , PHK allocator [44] , and the estranged buddy system [25] for both software and hardware implementations. I do not evaluate hardware complexities of these allocators, but rather simulate the existence of such allocators as special hardware units within the Simplescalar tools.

Lea's allocator, written by Doug Lea, is used in Linux systems. Lea's allocator is a best-fit allocator using lists of different-sized objects. For small objects, it uses exact-fit from the quick-lists by allocating an object from an appropriate-sized list; for medium and large-sized objects it uses best-fit. Coalescing of freed objects is performed as needed. The PHK allocator, used in FreeBSD systems, was designed by Poul-Henning Kamp. PHK allocators are page-based segregated allocators. Each memory page contains only objects of one size. For small objects (less than half a page) object size is padded to the nearest power of two. For larger objects, PHK will allocate the number of pages that is sufficient to satisfy the request. The page directory is allocated using mmap() system call. The estranged buddy system is a variation of Knuth's buddy system. In estranged buddy, buddies are not immediately combined into larger chunks, thereby eliminating the need for later breaking larger chunks into smaller ones.

I use the same simulation tool-SimpileScalar 3.0 for the simulation environment as in the previous chapter. I simulate memory management as a hardware functional unit residing on-chip along with the primary processing units. I show results for the memory functional units assumed to implement Doug Lea, PHK, and Estranged Buddy methods of memory management. The hardware allocator keeps all the object headers (i.e., bookkeeping data) and provides only actual data objects to the processing elements, thus improving the CPU data cache performance. As in chapter 6, I added new instructions to the SimpleScalar Portable Instruction Set Architecture (PISA), to perform malloc()/free()/realloc() functions of C.

The simulator configuration is same as that shown in Table 6.1. Table 7.1 describes the benchmarks used in this part of experiment. 253.perlmbk and 197.parser, from the

SPEC benchmarks suite, are the most memory-intensive in terms of the number of dynamic objects allocated; 255.vortex and 300.twolf are moderately intensive, and the remaining SPEC benchmarks have very few allocation requests. 176.gcc is not memory intensive in terms of using the system-provided allocator, since it maintains its own object stack allocator. cfrac and espresso are memory intensive.

It should be noted that system calls in SimpleScalar are assumed to take only one cycle. This may somewhat skew the results since SimpleScalar does not penalize allocators that invoke *brk()* system calls to get more memory from the system. However, in real implementations, systems calls can be expensive. Efficient memory management algorithms try to avoid *brk()*, using *mmap()* instead, or starting with a large memory space rather than incrementally asking the system for more space. Use of SimpleScalar does not allow us to differentiate allocators' use of system calls.

Benchmark Name	Benchmark Description	Input	Total Number of Allocated Objects
253.perlbmk	Perl interpreter	perfect.pl b 3 m 4	8,888,714
197.parser	Natural language processor	ref.in(100)	8,647,462
255.vortex	Object-oriented database	test/lendian.raw	186,429
300.twolf	Place and route simulator	test	8,395
164.gzip	gnu zip data compression	test/input.compressed 2	1,245
176.gcc	gnu C compiler	cccp.i -o cccp.s	4,304
175.vpr	FPGA circuit placement and routing	net.in place.in	1,590
181.mcf	Minimum cost network flow solver	test/inp.in	3
cfrac	Factors numbers	A 22-digit number	227,091
espresso	PLA optimizer	largest.espresso	1,701,112

TABLE 7.1. Selection of the Benchmarks, Inputs, and Number of Dynamic Objects

7.1. Performance of Different Algorithms

In this section I show that, when using hardware allocators, performance of applications is affected mostly by object localities, and that the actual allocator method used has less significant impact on applications' performance.

Table 7.2 shows the number of instructions executed by software and hardware implementations of three allocators - Doug Lea (DL), PHK, and Estrange Buddy (ESB). The leftmost columns list variation in numbers of instructions executed using different allocators implemented in software (including both allocator functions and application code). The rightmost columns show variations in instructions executed by the same set of allocators implemented in hardware. These numbers include only application code since allocator functions are not executed by the CPU.

Benchmark Name	# of Instructions Executed on Conventional Architecture			# of Instructions Executed on Separate Architecture		
	DL	PHK	ESB	DL	PHK	ESB
cfrac	124.9	238.35	152.8	95.8	95.8	95.8
197.parser	9,140	11,598	9,772	7,570	7,572	7,511
Espresso	2,091	2,363	2,072	1,863	1,863	1,863
253.perlbmk	30,415	35,039	31,345	29,545	29,545	29,546
255.vortex	13,092	13,130	13,051	12,995	12,995	12,995
164.gzip	4,557	4,558	4,557	4,557	4,557	4,557
300.twolf	412.9	411.9	414.2	411.9	411.9	411.9
175.vpr	2,372	2,373	2,373	2,372	2,372	2,372
176.gcc	2,526	2,526	2,526	2,524	2,524	2,524
181.mcf	418.9	420.2	418.9	418.9	418.9	418.9

TABLE 7.2. Number of Instructions(Million) Executed by Different Allocation Algorithms

Table 7.3 shows L-1 instruction cache miss behavior of the three allocators when implemented in software and when separated from the CPU. Unlike software implementations of the three allocators, hardware implementations do not show any variation in cache miss data, since allocator functions are not brought to the CPU instruction cache. In Table 7.4 I show L-1 data cache misses for applications using software and hardware implementations of the three different memory allocators (viz., Doug Lea, PHK, and Estrange Buddy system). Differences among cache behaviors of software implementations are both because the allocator data (viz. bookkeeping data) is brought into the CPU cache and because of differences in the localities of applications' objects. Differences in cache behaviors of hardware implementations are due only to differing localities of allocated objects. Even though hardware

allocators remove headers from the CPU data cache, the actual placement of object data depends on the allocator itself, affecting data localities and data cache misses.

Benchmark Name	# of L1-Instruction Cache Miss on Conventional Architecture			# of L1-Instruction Cache Miss on Separate Architecture		
	DL	PHK	ESB	DL	PHK	ESB
cfrac	6.74	9.87	6.40	4.21	4.21	4.21
197.parser	145	119	101	74.6	74.6	74.6
espresso	34.9	39.3	30.7	23.0	23.0	23.0
253.perlbmk	4,163	4,505	4,383	4,051	4,056	4,057
255.vortex	988	1018	1017	976	976	976
164.gzip	24.4	23.8	23.7	23.7	23.7	23.7
300.twolf	28.9	29.1	28.7	28.9	28.9	28.9
175.vpr	240	243	249	240	240	240
176.gcc	200	201	200	200	200	200
181.mcf	0.525	0.367	0.385	0.417	0.417	0.417

TABLE 7.3. Number of L1-Instruction Cache Misses(Million) of Different Allocation Algorithms

Benchmark Name	# of L1-Data Cache Miss on Conventional Architecture			# of L1-Data Cache Miss on Separate Architecture		
	DL	PHK	ESB	DL	PHK	ESB
cfrac	9998*	56,106*	16,805*	8,205*	13,721*	11,809*
197.parser	77.4	59.18	61.04	64.98	58.6	59.10
espresso	16.14	15.33	16.47	15.23	14.63	14.52
253.perlbmk	193.4	166.1	211.3	173.4	171.2	162.5
255.vortex	40.91	74.17	73.3	34.18	69.43	70.83
164.gzip	37.58	37.56	37.59	37.55	37.55	37.55
300.twolf	1.07	1.38	1.75	0.76	1.33	1.45
175.vpr	17.93	22.76	21.52	17.81	22.75	20.2
176.gcc	13.2	13.3	13.5	13.2	13.2	13.2
181.mcf	10.6	10.61	13.53	10.6	10.6	10.6
	*exact number					

TABLE 7.4. Number of L1-Data Cache Misses(Million) of Different Allocation Algorithms

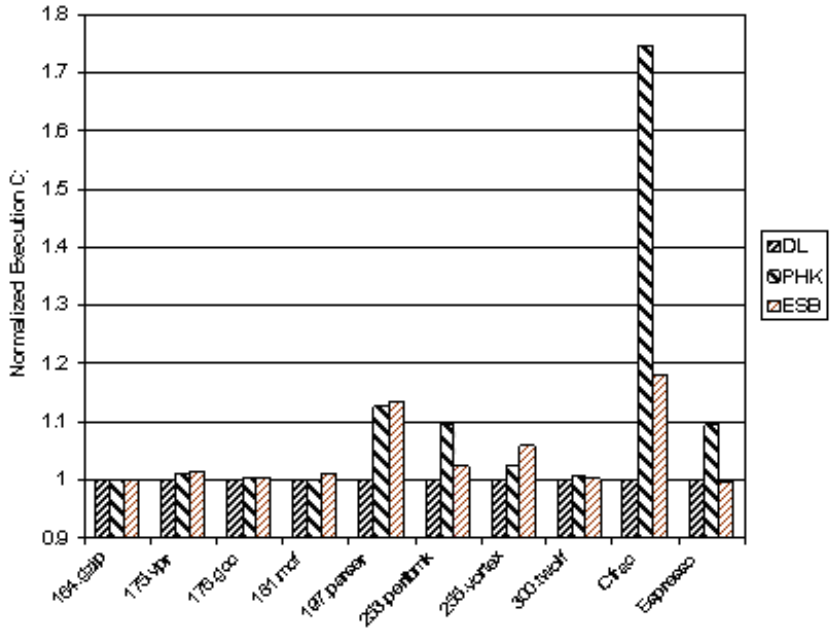
Both Tables 7.3 and 7.4 show reductions in instruction and data cache misses when using hardware for memory allocation (particularly for memory-intensive benchmarks). For example, for application cfrac, using Lea's allocator in hardware shows a reduction in instruction cache misses of 2.53 million and data cache misses of 1793. Likewise using PHK, hardware shows a reduction of 5.76 million in instruction cache misses, and 42385 in data cache misses. And the estranged buddy system shows reductions of 2.19 million and 4996 in instruction and data cache misses respectively.

Table 7.5 shows execution cycles for benchmarks using the three allocators when executed by the CPU and when executed by a separate hardware.

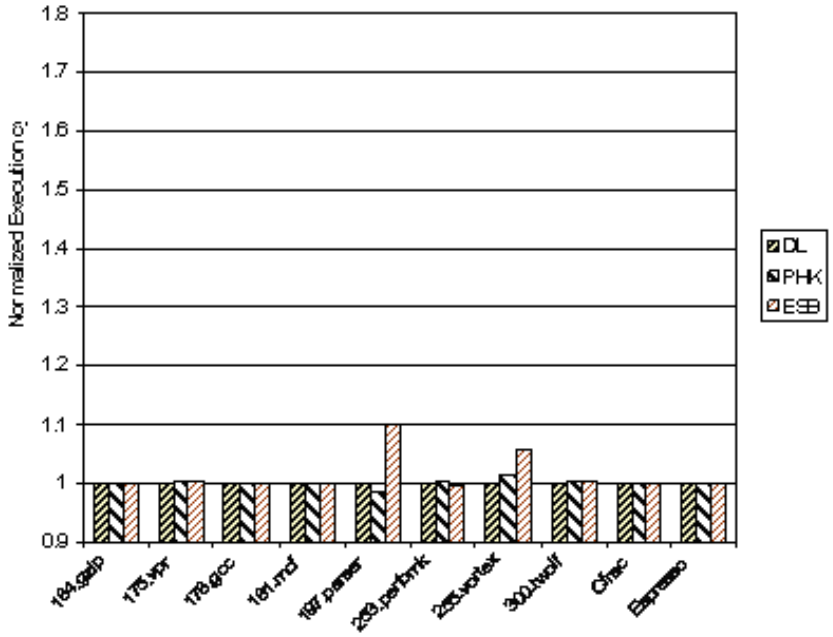
Benchmark Name	Conventional (million)			Separated (million)		
	DL	PHK	ESB	DL	PHK	ESB
cfrac	104.9	183.2	123.7	78.83	78.84	78.84
197.parser	7,348	8,277	8,345	5,996	5,901	6,605
espresso	1,379	1,511	1,377	1,202	1,201	1,202
253.perlbmk	45,801	50,179	46,872	43,981	44,120	43,810
255.vortex	12,807	13,122	13,560	12,553	12,728	13,287
164.gzip	2,539	2,536	2,536	2,507	2,507	2,508
300.twolf	433.3	436.8	434.1	432.5	434.4	434.1
175.vpr	3,136	3,168	3,188	3,141	3,153	3,155
176.gcc	2,663	2,671	2,669	2,661	2,661	2,661
181.mcf	309.6	309.6	312.4	309.6	309.1	309.1

TABLE 7.5. Execution Cycles of Three Allocators

In order to measure variations among the different allocators implemented in software and hardware, I present the data graphically. Figure 1(a) shows execution cycles using software implementations of the three allocators, normalized to execution cycles of the software's Doug Lea allocator. Figure 1(b) shows the execution cycles for the three allocators implemented in hardware, again normalized to the execution cycles of Doug Lea's allocator in hardware. The graphical data clearly shows greater variation in the execution cycles for applications when memory management functions are implemented in software, as compared to the case when the allocators are removed from the main processing elements (and implemented in hardware). I believe that this is a significant result indicating that one can



(a) Performance Using Software Allocators



(b) Performance Using Hardware Allocators

FIGURE 7.1. Performances of Hardware and Software Allocators

select an allocator that simplifies hardware implementation and improves localities of allocated objects without needing to worry about the impact of the allocator on the execution performance of applications.

In general, Lea's algorithm performs better than other allocators, both in hardware and software implementations. Lea's allocator utilizes both spatial and temporal locality in its allocation algorithm. But for benchmarks with more significant spatial localities, the PHK allocator sometimes exhibits better cache performance. This is the case with 197.parser and 253.perlbmk. Estranged buddy allocators show worse cache behavior because any buddy system allocator results in large internal fragmentations, and poor localities. However, performance differences when using hardware implementations are small. As stated previously, object locality depends on the allocator algorithm used, and application performance variations among the hardware allocators (Figure 1(b)) are mainly due to variations in L1 data cache localities exhibited by allocated data objects.

While using software-implemented allocators executed by main processing pipelines, not only the complexity of the allocator affects the performance of applications, but also the localities of allocated objects can lead to differences in applications' performance. However, when allocator functions are removed from the CPU, applications only see performance differences caused by localities of objects allocated. And the locality difference will not cause a significant performance impact, which suggests if implementing a hardware allocator, I should choose a simple and fast implementation.

CHAPTER 8

A HARDWARE/SOFTWARE CO-DESIGN OF A MEMORY ALLOCATOR

In the previous two chapters, I discussed the pure hardware implementation of a memory allocator completely separate from the execution pipeline. However, I did not show a hardware implementation. I only analyzed the performance benefit and the different algorithm impact on a hardware allocator.

In this chapter, I present a new hybrid software/hardware allocator and its hardware implementation for a faster, lower cost system. This allocator is based on the PHK [44] allocation algorithm used in the Free-BSD system and Chang's hardware design for allocators [14]. I aim to balance the hardware complexity with performance by using both hardware and software together. To substantiate the claims, I present a comparison of the design in terms of hardware complexity with a hardware-only allocator and a comparison in terms of performance with a software-only allocator. The proposed hybrid allocator can find important use in applications written in modern programming languages like C++/JAVA where a significant amount of time is spent in memory management.

Different software allocators use different techniques to organize available chunks of free memory. A search of these free chunks is needed for allocation of memory. This search could be in the critical path of allocators causing a major performance bottleneck. Hardware allocators can provide several advantages over their software counterparts. Parallel search of the available memory chunks can be implemented in hardware, which can speed up memory allocation and improve the performance by reducing execution time. The hardware allocator can easily hide the execution latency of freeing objects, since freeing can run concurrently with application execution. The hardware allocator unit can also perform coalescing of free chunks of memory, in the background, while the application is not using this portion of the memory. The major disadvantage of a hardware only allocator is the potential hardware complexity in implementing complex allocators.

Berger et al. [9] showed that a general purpose allocator works well for most applications. The average performance difference of the two most popular general purpose open source allocators, Doug Lea's [53] used in the LINUX system and PHK used in the Free-BSD system, is less than 3% for memory allocation intensive benchmarks in SPEC 2000. I chose PHK because of its suitability for hardware/software co-design. The PHK allocator is a page-based allocator. Each page can only contain objects of one size. For a large object, sufficient numbers of pages are allocated to accommodate the object. For applications written using object-oriented languages such as Java and C++, most of the objects allocated are small. For small objects, less than half a page, object size is padded to the nearest power of two, to match the size of objects on that page. This allocator keeps a page directory for all the allocated pages. At the beginning of each small object page, a bitmap of allocation information is created. When allocating a small object, the PHK allocator performs a linear search on the bitmap to find the first available chunk in that page. This search is performed in the following sequence: first locate the first word of a page that has a free chunk, then locate the address of the first byte in that word that represents a free chunk.

There are a few hardware allocator designs [14] [29] [25] [24] reported. All of these are based on the buddy system invented by Knuth [50]. Chang's algorithm [14] is a first-fit method based on a binary OR-tree and a binary AND-tree. Each leaf node of the OR-tree represents the base size of the smallest unit of memory that can be allocated, and other nodes provide information if such a unit is available. All allocated objects are multiples of the base size. The leaves of the OR-tree together represent the entire memory. The input of the AND-tree is generated by a complex interconnection network of the OR-tree. The AND-tree has the same number of leaves as the OR-tree. The AND-tree is used to generate the address of the first available chunk for a particular sized object. The interconnection between the OR-tree and the AND-tree is the most complex part of Chang's allocator. The interconnection has the same critical path delay as the OR-tree and the AND-tree. The final allocation result is produced by the output of the AND-tree through a set of multiplexers. The critical path delay of this algorithm is: $D_{delay} = D_{OR-tree} + D_{Interconnection} + D_{AND-tree}$.

The hardware complexity in terms of the number of gates is $O(nlg(n))$, where n is the number of the memory chunks and $O(ln(n))$ is the critical path delay.

8.1. The Proposed Hybrid Allocator

I note that pure hardware allocators based on the buddy system are not scalable since the complexity of the hardware increases with the size of the memory managed. Also, the buddy system is known for its poor object locality [43]. On the other hand, software allocators exhibit poor execution performance. I have designed a new hybrid allocator using small, fixed hardware to help manage the memory. The software portion based on the PHK algorithm provides better object localities than the buddy system and the hardware portion improves execution performance of the software portion.

The software in the allocator is responsible for creating page indexes and for initializing the page header as in a software implementation of PHK. For large objects (> half a page), the software takes full responsibility without any hardware assistance. When an application requests allocation of a small-sized object, the software portion of the hybrid system will locate the bitmap of a page with free memory and issues a search request to the hardware. The hardware portion will search the page index (or bitmap) in parallel to find a free chunk, and mark the bitmap to indicate an allocation.

Figure 8.1 shows the block diagram of the hardware I propose to fulfill parallel searching. I have an OR-tree and an AND-tree similar to Chang's system. The OR-tree is responsible for determining if there is a free chunk in a page. The AND-tree will locate the position of the first free chunk in the page. Because an OR-tree and an AND-tree are dedicated to one object size, the complex interconnections between the OR-tree and the AND-tree are not needed (unlike Chang's [14]). The individual implementation of the OR-tree and the AND-tree are identical to that of Chang's designs. The multiplexer (MUX) uses the opcode to select the address of the bit needed to be flipped. If the opcode is "alloc", the address from the AND-tree will be chosen. If the opcode is "free", the address from the request will be selected.

D-latches in the design are used as storage devices, where the bitmap will be loaded from the page in accordance with the allocation size. The de-multiplexer (DEMUX) is used to decode the address from the MUX. Bit-flippers use the decoded address and the opcode to determine how to flip a desired bit. Because of the page limits, I do not show the detailed

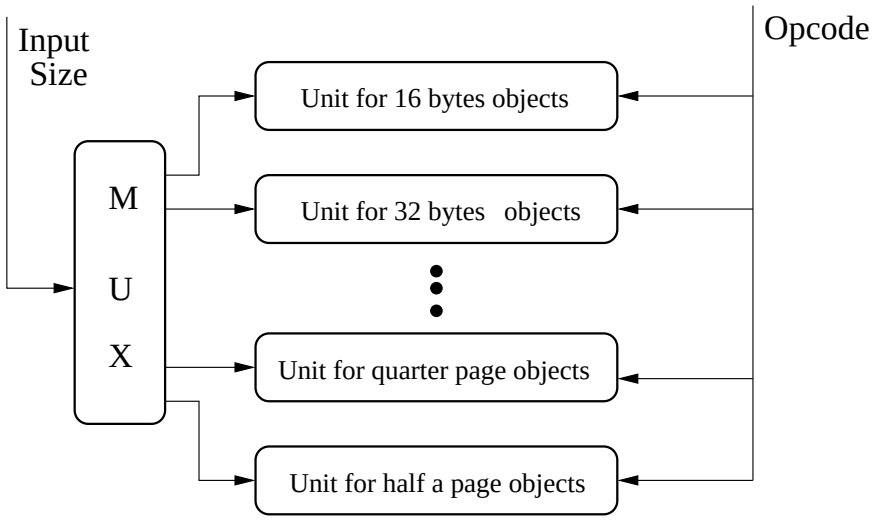


FIGURE 8.1. Block Diagram of Overall Hardware Design

flipper logic here. It may be noted that the critical path in this design is only the AND-tree for the "allocate" operation. The "free" doesn't generate any output, and the processor can immediately continue execution of the application code.

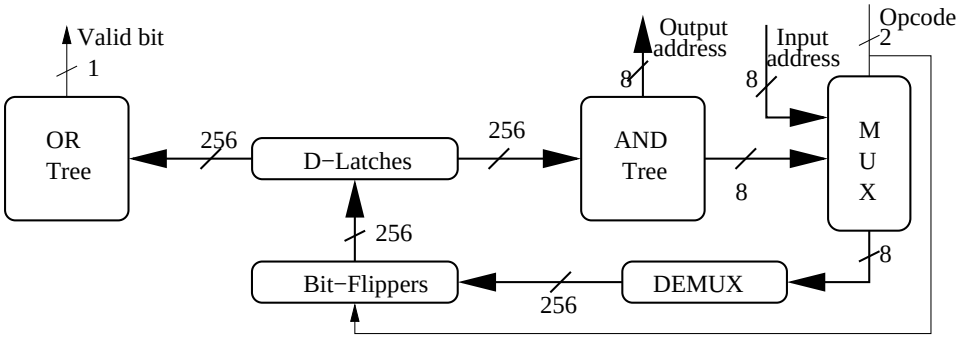


FIGURE 8.2. Block Diagram of the Proposed Hardware Component (Page Size 4096 bytes, Object Size 16 bytes)

Figure 8.1 shows the overall design of the system with 4096-byte pages. I have shown one unit for one page in Figure 8.2. For different object sizes, the hardware needed to support the bitmap will be different. In the design, I pre-selected object sizes from 16-bytes to 2048 bytes and included hardware to support pages for these objects. It should be noted that the larger the object size the smaller the amount of hardware needed to support the bit-maps indicating the availability of chunks in that page. For example, I need only 2 bits for a page that allocates 2048-byte objects. The MUX here is used to select the hardware unit that will be responsible for supporting objects of a given size. With 4096-byte pages, I have 8

Attributes	**Chang's Allocator**	**My Design**
Design Algorithm	Total Memory	Page Based
Interconnection Complexity	$O\left(\frac{M}{S} \lg \frac{M}{S}\right)$	No Interconnection
Overall Hardware Complexity	$O\left(\frac{M}{S} \lg \frac{M}{S}\right)$	$O\left(\frac{P}{S}\right)$
Scalability	No	Yes
Need for Software Assistance	No	yes
Critical Path Delay	$O\left(\lg \frac{M}{S}\right)$	$O\left(\lg \frac{P}{S}\right)$
Clock Frequency	Slow	Fast
Allocation Locality	Poor	Better
POSIX Compatible	No	Yes

TABLE 8.1. Comparison of Chang's Allocator and my Design

different-sized objects ranging from 16-bytes to 2048-bytes. For allocating 16-byte objects, I need trees with 256 leaves. Each tree only needs 255 AND/OR gates. For the overall system, I need 502 AND gates and 502 OR gates. This is very small amount of hardware compared with the billions of transistors available on modern processor chips.

8.2. Complexity and Performance Comparison

8.2.1. Complexity Comparison

Existing hardware allocator designs implement the buddy system of allocations. The amount of hardware that is used to implement a buddy allocator is dependent on the memory size [43]. That makes the buddy system-based allocators not scalable. The design has a much lower hardware complexity than Chang's allocator. In order to compare hardware complexity, the following notations are used: M is the total dynamic memory size, P is the page size, and S is the smallest allocated object size. Table 8.1 shows details of the comparison with Chang's algorithm.

The complex interconnection determines the hardware complexity of Chang's allocator and it grows as $O((M/S)lg(M/S))$. The hardware complexity of my design is $O(P/S)$. Normally, the page size is small and in most cases pages are of a fixed size. For example, in a 2GByte dynamic memory system where the smallest object allocated is 16-bytes, Chang's allocator needs several hundred million gates, while my design only needs twenty thousand gates when 4096-byte pages are used (see section 8.1).

The critical path delay of my design is much smaller than that of Chang's design. For Chang's allocator, the critical path delay is $O(lg(M/S))$ which grows with the size of the memory that is managed. For my design, the critical path delay is $O(lg(P/S))$. For a system as previously described, the height of the trees in Chang's algorithm is 27. The total critical path delay will be 108 logic gate delays. For my approach, the critical path incurs only 16 gate delays. Moreover, the proposed allocator can be run at a much higher clock frequency than Chang's allocator, although it needs software assistance.

When freeing an object, Chang's algorithm needs the size of the object to manipulate the AND-trees and OR-trees. In POSIX systems, "free" commands do not provide object sizes; only the starting address of the object to be freed. This incompatibility makes Chang's approach impractical. Since the software part in my design will locate the bitmap on free, my design is fully POSIX compatible. In addition, my design is based on the PHK allocator,which aims to enhance the locality of allocated objects (since smaller objects are allocated from the same page), unlike an allocator based on the buddy system used by Chang. However, there is another buddy allocator called the Address-Ordered buddy system [21] that may improve locality.

8.2.2. Performance Analysis

For the purpose of analyzing performance gains from my design, I simulated the existence of a hardware-assisted PHK allocator within a conventional CPU using a SimpleScalar simulation tool set. The hardware portion of the hybrid allocator presented in section 8.1 runs at 1-cycle speed. For the purpose of analysis this hardware is implemented as a special functional unit in a superscalar processor. This unit is activated by operations "find_chunk" and "free_chunk". The page size of the system is assumed to be 4096 bytes, and the smallest object allocated is set to 16 bytes. The detailed processor parameters used in the simulations is the same as in Table 6.1.

I used ten benchmarks (with varying numbers of memory management overheads) to study the performance gains using my design: parser and perlbmk are from SPEC CPU2000 suite; cfrac, espresso and boxed-sim are memory intensive benchmarks that are widely used by researchers; the other benchmarks are from the Olden suite, which are also memory

Benchmark Name	**Input**	**Average Object Size**	**Time Spent in Allocation** (%)
cfrac	22-digits number	8 bytes	29.7
espresso	largest.espresso	250 bytes	4.7
boxed-sim	-n 10 -s 1	24 bytes	2.4
parser	ref.in (first 300 lines)	16 bytes	35.6
perlbmk	perfect.pl b 2	38 bytes	10.7
treeadd	20 1	24 bytes	48.2
voronoi	20000 1	40 bytes	10.4
bisort	250000 1	24 bytes	2.3
perimeter	12 1	48 bytes	16.3
health	5 500 1	24 bytes	4.9

TABLE 8.2. Selected Benchmarks and Ave. Object Sizes

Benchmark Name	**PHK Software Allocator Execution Cycles (million)**	**My Hardware Allocator Execution Cycles (million)**	**Speedup**
cfrac	189.7	148.1	1.28
espresso	5,241	5,129	1.02
boxed-sim	9,043	8,922	1.01
parser	27,111	21,163	1.27
perlbmk	135.5	127.3	1.06
treeadd	160.4	112.4	1.43
voronoi	128.8	122.3	1.05
bisort	424.1	418.1	1.01
perimeter	42.11	37.97	1.11
health	383.0	372.2	1.03

TABLE 8.3. Performance Comparison with PHK Allocator

allocation intensive programs. The inputs to these benchmarks, average object sizes, and percentage of execution time spent in memory management are shown in Table 8.2. The simulation results are shown in Table 8.3.

The speedup of each application is proportional to the execution time spent on memory management and the average object size. In Figure 8.3, I show the reduced memory management execution cycles normalized to the original execution cycles spent on memory

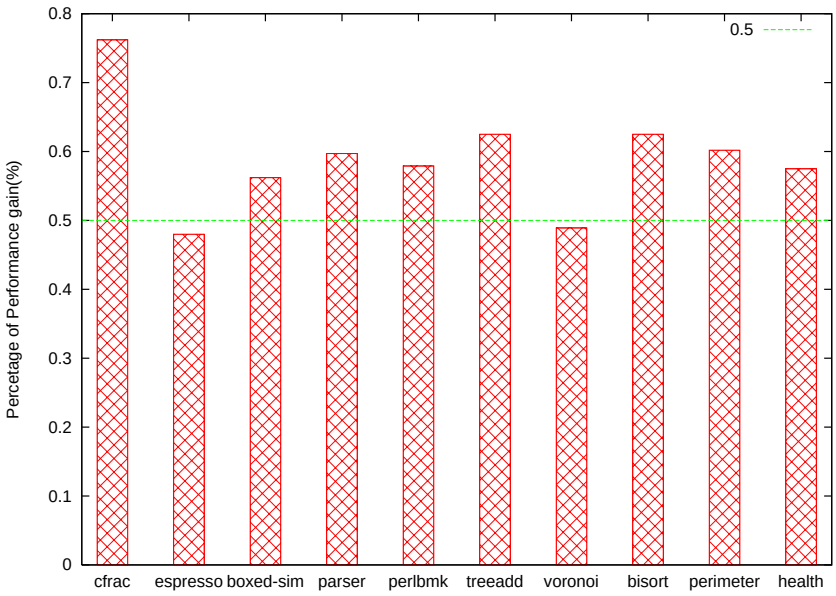


FIGURE 8.3. Normalized Memory Management Performance Improvement

management functions by the software only allocator. This figure shows the relative performance improvements for memory management functions. The cfrac application shows the best performance improvement. The average object size in cfrac is 8 bytes, which means that most pages allocated contain 256 objects. The linear search in the software implementation for that many objects will be very slow. The hardware speeds up the search, leading to a 76.2% normalized performance improvement over the software-only allocation. The cycles spent in bitmap searching by the software-only allocator is close to the performance difference between the software-only allocator and the proposed hybrid allocator, which can be calculated from Table 8.3.

The benchmark espresso with an average object size of 250 bytes shows the least amount of improvement using the hybrid allocator. Pages allocated for espresso contain fewer than 20 objects. Linear search of 20 objects is not significant, and the hardware allocator only shows a 48.0% normalized performance improvement. The other benchmarks have average object sizes of 16 bytes to 48 bytes, and thus the performance gains are not as significant as that for cfrac, but better than espresso.

On average, the hybrid allocator reduces the memory management time by 58.9%. The average overall execution speedup of the design when compared to a software-only allocator implementation is 1.127 (or 12.7%).

8.3. Conclusion

My design has significantly lower hardware complexity and lower critical path delays compared to reported hardware-only allocators. My hardware design has a fixed hardware complexity, complexity being dependent on the size of a memory page, and not the total (user) memory being managed. Since this design is based on a PHK algorithm, it is likely to achieve better object localities than those using buddy systems. I also have shown that the hardware-software allocator achieves 12.7% gains in overall execution performance over software-only allocator implementation for memory intensive benchmarks and improves the memory management efficiency by 58.9% (that is the execution performance improvement for memory management functions). The performance gains depend on how often an application invokes "malloc" or "free" functions, and the average size of objects allocated. In the future, I will explore variable-sized pages such that the number of allocated objects are the same in each page. By doing this, all the bitmaps will have the same number of bits. Thus, I will only need one pair of AND-trees and OR-trees in my design. That will further reduce the hardware complexity. I expect that this will also improve the memory management efficiency of allocators for large objects. I also plan to investigate hybrid designs for other memory management algorithms like Doug Lea's allocator.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1. Conclusions and Contributions

9.1.1. Contributions of TLS in dataflow architecture

This section will emphasize the contributions and conclusions of this dissertation in terms of designing, implementing, and evaluation of the Thread Level Speculation schema for multi-threaded dataflow architecture (SDF).

The major contribution of this dissertation is the design of a TLS schema that can be implemented for the SDF architecture. One major performance barrier of SDF architecture is that ambiguous data dependencies will force thread sequential execution. This TLS schema can enhance the performance of SDF architecture by eliminating the sequential execution due to static ambiguous data dependencies commonly existing in the programs. It will improve the utilization of the hardware by executing the instructions speculatively. As I have shown previously, part of the work, creating and initializing thread context, will always be useful in terms of SDF architecture. Modern compilers can easily capture these data dependencies and parallelize the code by utilizing the TLS hardware. A compiler extension based on the GCC 4.0 can already generate speculative threads for conventional architecture. And with the trend of multi-core architectures becoming mainstream architecture and the revival of dataflow concepts, thread-level speculation will become an increasingly popular research topic.

The second contribution is that I demonstrated a hardware implementation of the TLS schema with simple hardware. The hardware used to implement this schema is a slight modification of a fully-associative cache to store the states of speculative access data. The SDF feature of preload and post-store makes the TLS scheme simpler than existing speculation support for conventional architectures [68] [31] [57] [33] [81].

I evaluated the performance of the proposed schema and show that even with a very high mis-speculation rate above 80%, TLS-SDF architecture can still achieve performance improvements. I also show that this scheme can scale better in terms of added functional units.

9.1.2. Contributions of hardware memory management

The major contribution in this subject is that I provide an analysis of the performance impact of using hardware memory management unit in conventional architectures. I applied different memory management algorithm in conventional architectures and showed that a simple memory management algorithm can achieve comparable performance as a complex one if implemented in hardware.

The second contribution is that I provide a hybrid (hardware/software) co-design allocator. I show that with this design, only a fixed amount of hardware (and the hardware does not grow with the total amount of memory managed) is needed. This implementation achieves an average of 12.7% performance gain.

9.2. Future Work

All the benchmarks used in evaluating the TLS SDF architecture are handwritten. The implementation of a compiler which supports the TLS execution in SDF needs to be addressed. The existing compiler framework - GCC, SUIF, and SCALE - can perform very complicated data dependency analysis. Based on the results of these analyses, one can easily generate the speculative threads code.

Lepek and Lipsti [54] mention that up to 70% of stores are silent in SPEC95, which means that the updated value is the same as the value already stored in that memory location. In the current TLS schema, these silent stores are treated as storing a new value, which will trigger invalidation of current cache copies. Detecting the silent stores and converting them to no-ops will eliminate some cache invalidations and improve the success rate of speculative threads. This will further enhance the performance.

As mentioned in chapter 5, the current simulator does not implement the multiple clusters of SDF nodes. Testing the TLS schema in a multiple cluster environment will be a project to undertake in the future. There are other challenges in a multiple cluster environment, such

as how this TLS schema can be integrated with the task scheduler to achieve the optimal performance.

In term of hardware memory management, I can extend the hardware memory management to include garbage collection. I believe that hardware memory management unit will show more performance advantage in garbage collection than its software counterpart.

BIBLIOGRAPHY

[1] W.B. Ackermann and J.B. Dennis. VAL – a value-oriented algorithmic language, preliminary reference manual. Technical Report TR218, Laboratory for Computer Science, MIT, Cambridge, MA, 1992.

[2] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*, pages 248–259, 2000.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, 1990.

[4] B.S. Ang, Arvind, and D. Chiou. StarT - the next generation: Integrated global caches and dataflow architecture. Tech Report TR-354, Laboratory for Computer Science, MIT, Cambridge, MA, August 1988.

[5] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[6] A.R.Hurson, J.T.Lim, K.M.Kavi, and B.Lee. Parallelization of do all and do across loops - a survey. *Advances in Computers*, 45:53–103, 1997.

[7] Arvind and R. Nikhil. Executing program on the MIT Tagged-token dataflow architecture. *IEEE Transactions on computers*, 39(3):300–318, 1991.

[8] Arvind and R.S. Nikhil. Executing a program on the MIT Tagged-token dataflow architecture. *PARLE(2)*, pages 1–29, 1987.

[9] E.D. Berger, B.G. Zorn, and K.S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–12, 2002.

[10] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[11] A.P.W. Böhm, D.C. Cann, J.T. Feo, and R.R. Oldehoeft. SISAL reference manual (language version 2.0). Technical Report CS91-118, Computer Science Department, Colorado State University, 1992.

[12] W.C. Brantley, K.P. McAuliffe, and J. Weiss. Rp3 processor-memory element. In *Proceedings of the International Conference on Parallel Processing*, pages 782–789, 1985.

[13] D. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Tech. Report CS-1342, University of Wisconsin-Madison, June 1997.

[14] J.M. Chang and E.F. Gehringer. A high-performance memory allocator for object-oriented systems. *IEEE Transactions on Computers*, 45(3):357–366, 1996.

[15] P.S. Chen, M.Y. Hung, Y.S. Hwang, R.D. Ju, and J.K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 25–36, June 2003.

[16] M. Cintra, J.F. Martnez, and J. Torrellas. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pages 13–24, June 2000.

[17] HP Compiler and Architecture Research Group. Trimaran, an infrastructure for research in instruction-level parallelism, `http://www.trimaran.org`.

[18] D.E. Culler, K.E. Schauser A.Sah, T. Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 164–175, April 1991.

[19] D.E. Culler, S.C. Goldstein, K.E. Schauser, and T.V. Eicken. TAM - a compiler controlled thread abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, 1993.

[20] D.E. Culler and G.M. Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[21] D.C. Defoe, S.R. Cholleti, and R.K. Cytron. Upper bound for defragment buddy heaps. In *Proceedings of Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 222–229, 2005.

[22] Jack B. Dennis and David Misunas. A preliminary architecture for a basic data flow processor. In *Proceedings of 2nd International Conference on Computer Architecture (ISCA-2)*, pages 126–132, January 1975.

[23] D.F.Snelling. *The Stateless Data-Flow Architecture*. PhD thesis, Dept. Comp. Sci., Univ. Manchester, 1993.

[24] S. Donahue, M. Hampton, R. Cytron, M. Franklin, and K. Kavi. Hardware support for fast and bounded-time storage allocation. In *Second Workshop on Memory Perfoamnce Issue (WMPI 2002)*, 2002.

[25] S. Donahue, M. Hampton, M. Deters, J.M. Nye, R. Cytron, and K. Kavi. Storage allocation for real-time, embedded systems. In *Embedded Software: Proceedings of the First International Workshop (EMSOFT)*, pages 131–147, October 2001.

[26] Z.H. Du, C.C. Lim, X.F. Li, C. Yang, Q. Hao, and T.F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *In Proceedings of the Conference on Programming Language, Design and Implementation (PLDI)*, June 2004.

[27] J. Edler, A. Gottlieb, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, M. Snir, P.J. Teller, and J. Wilson. Issues related to mimd shared-memory computers: The nyu ultra-computer approach. In *Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA-12)*, pages 126–135, June 1985.

[28] D. Patterson et al. The case for intelligent RAM:IRAM. *IEEE Micro*, pages 34–44, April 1997.

[29] H. Cam et al. A high performance hardware efficient memory allocator technique and design. pages 274–276, 1999.

[30] S.J. Frank. Tightly coupled multiprocessor systems speed memory access times. *Electronics*, 57(1):164–169, January 1984.

[31] M. Franklin and G. S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, pages 552–571, May 1996.

[32] K.M. Kaviand R. Giorgi and J. Arul. Scheduled dataflow: Execution paradigm, architecture and performance evaluation. *IEEE Transactions on Computers*, 50(8):834–846, August 2001.

[33] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[34] V.G. Grafe and J.E. Hoch. The Epsilon-2 multiprocessor system. *Journal of Parallel and Distributed Computing*, 10:309–318, 1990.

[35] W. Grünewald and T. Ungerer. A multithreaded processor design for distributed shared memory system. In *Proceedings of the International Conference on Advances in Parallel and Distributed Computing*, pages 206–213, 1997.

[36] J.R. Gurd and D.F.Snelling. Manchester data-flow: a progress report. In *Proceedings of the 6th international conference on Supercomputing*, pages 216–225, 1992.

[37] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[38] S. Heller and T. Ungerer. Id compiler user's manual. Technical Report MIT/CSG Memo 248, Laboratory for Computer Science, MIT, Cambridge, MA, 1985.

[39] S. Hily and A. Seznec. Contention on 2nd level cache may limit the effectiveness of smt. Internal Report 1086, IRISA, 1997.

[40] S. Hily and A. Seznec. Out of order execution may not be cost-effective on processors featuring smt. Internal Report 1179, IRISA, 1998.

[41] R.A. Iannuucci. Toward a dataflow/von neumann hybrid architecture. In *Proceedings of the 15th International Symposium on Computer Architecture (ISCA-15)*, pages 131–140, 1988.

[42] T. Johnson, R. Eigenmann, and T. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *In Proceedings of the Conference on Programming Language, Design and Implementation (PLDI)*, June 2004.

[43] M.S. Johnstone and P.R. Wilson. The memory fragmentation problem: solved. In *ISMM'98 Proceedings of the First International Symposium on Memory Management, vol. 34(3), of ACM SIGPLAN Notices*, pages 26–36, October 1998.

[44] P.H. Kamp. Malloc(3) revisited, `http://phk.freebsd.sk/pubs/malloc.pdf`.

[45] R. Katz, S. Eggers, D.A. Wood, C. Perkins, and R.G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA-12)*, pages 276–283, June 1985.

[46] K.M. Kavi, J. Arul, and R. Giorgi. Execution and cache performance of the scheduled dataflow architecture. *Journal of Universal Computer Science, Special Issue on Multithreaded and Chip Multiprocessors*, October 2000.

[47] K.M. Kavi and A.R. Hurson. Performance of cache memories in dataflow architectures. *Journal of System Architecture*, 44(9-10):657–674, June 1998.

[48] K.M. Kavi, B. Lee, and A.R. Hurson. Multithreaded systems: A survery. *Advances in Computers*, 48:287–328, 1998.

[49] M. Kishi, H. Yasuhara, and Y. Kawamura. DDDP – a distributed data driven processor. In *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA-10)*, pages 236–242, June 1983.

[50] D.E. Knuth. *The Art of Computer Programming* ***Vol I****: Fundamental Algorithms.* Addison-Wesley, 1968.

[51] I. Koren, B. Mendelson, I. Peled, and G.M. Silberman. A data-driven vlsi array for arbitrary algorithms. *Computer*, 21:30–43, October 1988.

[52] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers (Special Issue on Multithreaded Architecture)*, pages 866–880, December 1999.

[53] D. Lea. A memory allocator, `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[54] K. Lepak and M. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, June 2000.

[55] W. Liu, J. Tuck, L. Ceze, K. Strauss, J. Renau, and J. Torrellas. A tls compiler that exploits program structure. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 2006.

[56] D. Madon, E. Sanchez, and S. Monnier. A study of simultaneous multithreaded architecture. In *Proceedings of EuroPar '99, LNCS vol. 1685, Springer-Verlag*, pages 716–726, September 1999.

[57] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *Proceedings of the International Conference on Supercomputing*, pages 77–84, July 1998.

[58] E. McCreight. The dragon computer system: An early overview. Technical report, Xerox Corp., September 1984.

[59] R.S. Nikhil and Arvind. Can dataflow subsume von neumann computing? In *Proceedings of the 16th International Symposium on Computer Architecture (ISCA-16)*, pages 262–272, 1989.

[60] G.M. Papadopoulos. Implementation of a general purpose dataflow multiprocessor. Tech Report 432, Laboratory for Computer Science, MIT, Cambridge, MA, August 1988.

[61] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memory. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA-11)*, pages 348–354, June 1984.

[62] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *ACM International Conference on Supercomputing (ICS)*, June 2005.

[63] M. Rezaei and K.M. Kavi. Intelligent memory management eliminates cache pollution due to memory management functions. *Journal of Systems Architecture*, 52(1):41–55, January 2006.

[64] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C.K. Kim, D. Burger, S.W. Keckler, and C.R. Moore. Exploiting ilp, tlp, and dlp using polymorphism in the trips architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-30)*, pages 422–433, June 2003.

[65] J.E. Smith. Decoupled access/execute architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[66] M. Tokoro snd R. Jagnanathan and H. Sunahara. On the working set concept for dataflow machine. In *Proceedings of the 10th International Symposium on Computer Architecture (ISCA-10)*, pages 90–97, July 1987.

[67] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA-22)*, pages 414–425, 1995.

[68] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pages 1–12, June 2000.

[69] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *The 36th Annual International Symposium on Microarchitecture (MICRO-36)*, 2003.

[70] M. Takesue. A unified resource management and execution control mechanism for dataflow machine. In *Proceedings of 14th Intl. Symposium on Computer Architecture (ISCA-14)*, pages 90–97, June 1987.

[71] C.K. Tang. In *National Computer Conference (AFIPS)*, pages 749–753, 1976.

[72] S.A. Threson and A.N. Long. A feasibility study of a memory hierarchy in dataflow architecture. In *Proceedings of the International Conference on Parallel Processing*, pages 356–360, June 1987.

[73] J.Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.C. Yew. The superthreaded processor architecture. *IEEE Trans. on Computers*, 48(9):881–902, September 1999.

[74] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA-22).*

[75] A.V. Veidenbaum. A compiler-assisted cache coherence solution for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 1029–1036, 1986.

[76] T. Vijaykumar and G. Sohi. In *International Symposium on Microarchitecture*, pages 81–92, November 1998.

[77] I. Watson and J.R. Gurd. A prototype dataflow computer with token labeling. In *Proceedings of the National Computer conference(AFIPS Proceedings 48)*, pages 623–628, 1979.

[78] P.R. Wilson, M.S. Johnstone, M.Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science 985*, 1995.

[79] W.C. Yen, D.W.L. Yen, and K.S. Fu. Data coherence problem in a multicache system. *IEEE Transaction on Computers*, 34(1):56–65, 1985.

[80] A. Zhai, C. Colohan, J. Steffan, and T.Mowry. Compiler optimization of scalar value communication between speculative threads. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[81] Y. Zhang, L. Rauchwerger, and J. Torrelas. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA-5)*, pages 135–141, January 1999.