EVALUATING THE SCALABILITY OF SDF SINGLE-CHIP MULTIPROCESSOR
ARCHITECTURE USING AUTOMATICALLY PARALLELIZING CODE

Yuhua Zhang, B.E.

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2004

APPROVED:

Krishna Kavi, Major Professor and Chair of the
Department of Computer Science and Engineering
Phil Sweany, Committee Member
Robert Brazile, Committee Member and Chair of
Graduate Studies in the Department of
Computer Science and Engineering
Oscar Garcia, Dean of College of Engineering
Sandra L. Terrell, Dean of the Robert B. Toulouse
School of Graduate Studies
Advances in integrated circuit technology continue to provide more and more transistors on a chip. Computer architects are faced with the challenge of finding the best way to translate these resources into high performance. The challenge in the design of next generation CPU (central processing unit) lies not on trying to use up the silicon area, but on finding smart ways to make use of the wealth of transistors now available. In addition, the next generation architecture should offer high throughout performance, scalability, modularity, and low energy consumption, instead of an architecture that is suitable for only one class of applications or users, or only emphasize faster clock rate. A program exhibits different types of parallelism: instruction level parallelism (ILP), thread level parallelism (TLP), or data level parallelism (DLP). Likewise, architectures can be designed to exploit one or more of these types of parallelism. It is generally not possible to design architectures that can take advantage of all three types of parallelism without using very complex hardware structures and complex compiler optimizations. We present the state-of-art architecture SDF (scheduled data flowed) which explores the TLP parallelism as much as that is supplied by that application. We implement a SDF single-chip multiprocessor constructed from simpler processors and execute the automatically parallelizing application on the single-chip multiprocessor. SDF has many desirable features such as high throughput, scalability, and low power consumption, which meet the requirements of the next generation of CPU design. Compared with superscalar, VLIW (very long instruction word), and SMT (simultaneous multithreading), the experiment results show that for application with very little parallelism SDF is comparable to other architectures, for applications with large amounts of parallelism SDF outperforms other architectures.
ACKNOWLEDGMENTS

This thesis would not have been possible without the help and support of my advisor Dr. Krishna Kavi. I would also like to thank my committee members Dr. Phil Sweany and Dr. Robert Brazile for reviewing and serving on the committee. Two other people deserved thanks for their work with this thesis. Litong Song wrote code for the front-end of SDF compiler. Wentong Li wrote parts of code generation and optimization. Finally, thanks for supports from my wife Feili Qin and family.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vi</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Related Work</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Organization</td>
<td>7</td>
</tr>
<tr>
<td>2. SDF MACHINE MODEL</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Decoupling Memory Access from Execution Pipeline</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Dataflow Model and Architectures</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 The Scheduled Dataflow Processor</td>
<td>10</td>
</tr>
<tr>
<td>3. SUIF INTERFACE</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Overview of the SUIF1.0 Hierarchy</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Intermediate Format of SUIF</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Automatic Parallelization Supported by the Infrastructure of SUIF</td>
<td>15</td>
</tr>
<tr>
<td>4. FRONT END AND CODE GENERATION OF COMPILER</td>
<td>18</td>
</tr>
<tr>
<td>4.1 Front End</td>
<td>19</td>
</tr>
<tr>
<td>4.1.1 Creating SDF Threads</td>
<td>19</td>
</tr>
<tr>
<td>4.1.2 Thread-Level Speculative Execution</td>
<td>21</td>
</tr>
<tr>
<td>4.1.3 Loop Transformations</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Code Generation</td>
<td>25</td>
</tr>
<tr>
<td>4.2.1 Thread Generation</td>
<td>25</td>
</tr>
<tr>
<td>4.2.2 Cascaded If-statement Optimization</td>
<td>26</td>
</tr>
<tr>
<td>4.2.3 Constant Propagation</td>
<td>27</td>
</tr>
<tr>
<td>5. RESULTS AND DISCUSSIONS</td>
<td>28</td>
</tr>
<tr>
<td>5.1 Methodology</td>
<td>28</td>
</tr>
<tr>
<td>5.2 Experimental Results for superscalar Architecture</td>
<td>29</td>
</tr>
<tr>
<td>5.3 Experimental Results for VLIW Architecture</td>
<td>32</td>
</tr>
</tbody>
</table>
5.4 Experimental Results for SMT Architecture ........................................... 34
5.5 Experiments for Effect of Parallelizing Level on Performance .......... 35

6. CONCLUSIONS .................................................................................................................. 39

REFERENCES ......................................................................................................................... 40


LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 5-1</td>
<td>SDF vs. superscalar Using Varying Number of Functional Units</td>
<td>30</td>
</tr>
<tr>
<td>Table 5-2</td>
<td>SDF vs. VLIW Using Varying Number of Functional Units</td>
<td>33</td>
</tr>
<tr>
<td>Table 5-3</td>
<td>SDF vs. SMT by Executing Multiple Programs in Parallel</td>
<td>35</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Thread Continuation Transitions Handled by the Scheduling Unit (SU)</td>
<td>12</td>
</tr>
<tr>
<td>3-1</td>
<td>Class Hierarchy of SUIF 1.0 Data Structures</td>
<td>14</td>
</tr>
<tr>
<td>3-2</td>
<td>Intermediate Representation of SUIF</td>
<td>15</td>
</tr>
<tr>
<td>3-3</td>
<td>Infrastructure of SDF Compiler</td>
<td>16</td>
</tr>
<tr>
<td>4-1</td>
<td>Infrastructure of SDF Automatically Parallelizing Compiler</td>
<td>19</td>
</tr>
<tr>
<td>4-2</td>
<td>Illustration of SDF Thread Fragment Code</td>
<td>21</td>
</tr>
<tr>
<td>4-3</td>
<td>Illustration of SDF Thread Speculative Execution</td>
<td>23</td>
</tr>
<tr>
<td>4-4</td>
<td>Illustration of Loop Transformation</td>
<td>24</td>
</tr>
<tr>
<td>4-5</td>
<td>Privatization and Reduction Transformation</td>
<td>25</td>
</tr>
<tr>
<td>4-6</td>
<td>Simplified Illustration of Cascaded If-statement</td>
<td>27</td>
</tr>
<tr>
<td>5-1</td>
<td>Performance Comparison of SDF and superscalar Architecture</td>
<td>31</td>
</tr>
<tr>
<td>5-2</td>
<td>IPC comparison of SDF and superscalar Architecture</td>
<td>32</td>
</tr>
<tr>
<td>5-3</td>
<td>Performance Comparison of SDF and VLIW Architecture</td>
<td>33</td>
</tr>
<tr>
<td>5-4</td>
<td>IPC Comparison of SDF and VLIW Architecture</td>
<td>34</td>
</tr>
<tr>
<td>5-5</td>
<td>IPC Comparison of SDF and SMT Architecture</td>
<td>35</td>
</tr>
<tr>
<td>5-6</td>
<td>Evaluating Effect of Different Parallelizing Level of Matrix Multiplication</td>
<td>37</td>
</tr>
<tr>
<td>5-7</td>
<td>Evaluating Effect of Different Parallelizing Level of Zoom</td>
<td>38</td>
</tr>
<tr>
<td>5-8</td>
<td>Evaluating Effect of Different Parallelizing Level of Jpeg</td>
<td>44</td>
</tr>
<tr>
<td>5-9</td>
<td>Evaluating Effect of Different Parallelizing Level of Autocorrelation</td>
<td>44</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

1.1 Overview

The performance gap between processors and memory has widened in recent years and this trend appears to continue in the future. In addition, 50-70% of the time of modern processors is stalled on memory loads due to latency. Memory latency has become the bottleneck to achieve high performance. Processor designers employ different methods to alleviate the effect of memory latency on high performance. For example, superscalar architecture uses complicated hardware techniques to dynamically schedule instructions to overlap the execution of long latency instructions while VLIW (very long instruction word) architecture uses sophisticated compiler techniques to statically pack independent instructions into one packet to improve ILP (instruction level parallelism). SMT (simultaneous multithreading) architecture uses multiple threads running simultaneously to tolerate the memory latency by replicating functional units in the superscalar architecture. These methods are limited in one way or another. Therefore, it is important to consider other possible ways for future processor. The SDF (scheduled data flow) architecture presented in this thesis can eliminate the effect of memory latency and achieve high performance by using multithreading and decoupled memory access techniques [1].

General-purpose microprocessors owe their success to their abilities to run many diverse workloads very well. Nowadays, most processors are not general-purpose, but serve for special applications such as desktop, network, server, scientific, graphic, and digital signal processors, which match the particular parallelism characteristics of their application domains [2]. The architectures with specifically oriented purposes have
limited scalability. On the contrary, a single-chip multiprocessor architecture designed for
general purposes displays great scalability.

During the 1990s, the design of CPU (Central Processing Unit) was focused on
boosting ILP. For example, complicated hardware techniques were used in the
superscalar architecture to explore ILP of the applications and higher clock rate was
required to implement longer pipeline stages. The complicated processor structure limited
the scalability of superscalar. In addition, the use of high clock rate dramatically
increased the power consumption. Therefore we have to reverse the trend toward
evermore complex and less power-efficient cores by simplifying the processor core [5].

SDF architecture that is based on dataflow eliminates the need of runtime
instruction scheduling and requires a simple pipeline design. Each processor element in a
SDF single-chip multiprocessor architecture is simple in design, compared to the modern
processor cores such as superscalar and SMT. Simple cores are more efficient in terms of
performance per amount of silicon, or watt of power. Provided that the cache and
memory system can keep up, the performance of an architecture with multiple simple
cores would generally scale better for multi-threaded code. The design of SDF
architecture is based on this philosophy. Multiple SDF processor elements built on a
single die can execute multiple instruction streams from different threads concurrently
and independently. In a SDF single-chip multiprocessor architecture, the speed of CPU is
reduced; however, the throughput of workload is increased and the scalable performance
is achieved. In addition, most of the execution cycles in SDF computation model are
useful. In traditional architectures extracting ILP, cycles are wasted to implement branch
prediction and instruction-level speculative execution. These techniques were not
implemented in current SDF architecture. Therefore, the SDF program diagram provides an efficient computation model. The reason few significant improvements on scalability in the complex single core architecture is lack of an efficient computational model. In addition, cycles lost due to speculative execution in superscalar and cache misses produced by the out-of-order execution defect the benefits of out-of-order execution in the superscalar architecture.

Three levels of parallelism, ILP (instruction level parallelism), TLP (thread level parallelism), and DLP (data level parallelism), are available in applications. There does not exist any computer architecture without complex hardware which can explore all three levels of parallelism. However, the next generation computer architectures should make most of the three level parallelisms, maximize their use of the available resources, and avoid escalating complex and non-scalable hardware structures. Most of the designs of the conventional computer architectures have only focused on the ILP. In fact, for most applications, high performance can be obtained by exploiting TLP of these applications. Adding a second thread to an existing application may double the performance. In other words, the TLP significantly increases the throughput of applications, compared to the conventional techniques exploring the ILP. In order to maximize the TLP of an application, a single-chip multiprocessor executes multiple threads in parallel across multiple processor cores in a single die. Exploiting the coarser-grained TLP in a single-chip multiprocessor has the same meaning as a wide-issue processor attempting to exploit the fine-grained ILP within a single thread [18].

To explore TLP, the application has to be partitioned into multiple threads that can be executed in parallel on different cores of a single-chip multiprocessor. The task of
partitioning programs do not have to be imposed on programmers since writing the parallel version of a program which both behaves correctly and achieves good performance is notoriously difficult. A compiler should automatically convert a program into its parallel version that retains correct semantics of the program and has efficient performance [22]. In our research, the SDF compiler based on the SUIF (Stanford University Intermediate Format) infrastructure can convert a conventional program into its parallel version. We compiled four benchmarks using the SDF compiler and executed these benchmarks on the SDF architecture simulator. By comparisons, the SDF architecture exhibits more scalability in performance than other modern architectures.

1.2 Related Work

Several architectures are available for architecture research. Each has its own limitation. Traditional computer architectures gain high performance by extracting the ILP of applications. For example, superscalar architecture uses complex hardware to evaluate the data dependencies of the instructions and to schedule instructions on the functional units at run time. VLIW architecture depends entirely on a compiler to analyze the dependencies of instructions and to extract the ILP of applications at compile time. Dynamic instruction scheduling techniques used in superscalar significantly increase the hardware complexity. Although no dynamic instruction scheduling techniques are used in the VLIW architecture, VLIW architecture requires more sophisticated compilers to extract ILP prior to execution [7].

Compared to traditional architectures extracting ILP, multithreading architectures, which support the concurrent execution of multiple threads, achieve high performances by exploiting TLP. Multiple sets of registers for keeping different thread states are used
in a multithreading processor to execute multiple threads concurrently. At present, there are several kinds of multithreading architectures on the market, namely, the Intel HyperThreading as well as the IBM POWER4 and Sun MAJC-5200 Dual-Core. There are two strategies to implement multithreading techniques: fine-grained multithreading and coarse-grained multithreading. On any given cycle, a fine-grained multithreading processor executes an instruction from one of the thread contexts which are kept on the register files. On the next cycle, it switches to a different thread context and executes an instruction from the new thread context. There is a context switch in a fine-grained processor on any given cycle. Unlike fine-grained processor, a coarse-grained processor may or may not have a context switch on any given cycle, which depends on the ILP of a thread. If a thread has a higher instruction-level parallelism, instructions issued from that thread will be filled all available functional units for the current execution; threads have lower instruction-level parallelism, instructions are interleaved from all these threads [4].

Our SDF architecture presents novel multithreading techniques that eliminate the memory stalls of a processor and achieves multithreading performance using a new program paradigm, which is based on decoupled memory access and non-blocking threads. A SDF processor contains two separate processing elements: the synchronization processor (SP) and the execution processor (EP). Unlike superscalar, our architecture does not perform out-of-order execution and thus eliminates the need for complex instruction issue and retiring hardware [3]. Compared to the instruction-level speculative execution in VLIW architecture, our architecture can speculatively execute applications in the thread level. In fact, speculative thread-level execution has more significant potential to boost performance than speculative instruction-level execution, since
exploiting coarse-grain parallelism is more efficient than extracting fine-grain parallelism. SMT architecture requires a complex instruction fetch unit to keep functional units busy and imposes higher pressure on the memory bandwidth. The complex instruction fetch logic required a significant amount of silicon area. Higher demand on memory bandwidth consumes more power. On the contrary, SDF architecture eliminates those requirements that limit SMT architecture from achieving high performance, since each core of SDF is a simple pipeline.

SimpleScalar\(^\text{1}\) [29] is a popular simulation environment for the computer architecture research. It provides several libraries which can be modified easily to perform and simulate the superscalar architectures. SimpleScalar supports different combined models of instruction issue and execution, for example, in-order issue and out-of-order execution, or out-of-order issue and out-of-order execution. In addition, SimpleScalar allows us to configure different parameters for different experimental environments, such as functional units, cache size, issue rate, issue window width and so on. Trimaran\(^\text{2}\) simulator is a research compiler focused on the VLIW architectures. It integrates the compilation and simulation together by a graphical interface. Ssmt-1.0 is a linux-based system which simulates the execution of SMT architecture by extending the out-of-order execution of SimpleScalar simulator [20]. There does not exist a compiler for SMT architecture. Hence, Ssmt-1.0\(^\text{3}\) simulator uses the utilities of SimpleScalar simulator to compile programs and simulates multithreading environment by running multiple

---

\(^{1}\) SimpleScalar© SimpleScalar, LLC, available at [www.simplescalar.com](http://www.simplescalar.com).

\(^{2}\) Trimaran simulator available at [www.trimaran.org](http://www.trimaran.org).

\(^{3}\) Ssmt-1.0 simulator available at [http://maggini.eng.umd.edu/vortex/ssmt.html](http://maggini.eng.umd.edu/vortex/ssmt.html).
contexts simultaneously whereby each context contains the execution of an application. Using SDF cycle-level simulator and the tools above mentioned, we have compared the execution performances of our architecture with those of superscalar, VLIW, and SMT architectures with multiple functional units.

1.3 Organization

This thesis explores the evaluation of scalability of SDF single-chip multiprocessor architecture using automatically parallelizing code. In Chapter 2, a general machine model of SDF architecture is described. Chapter 3 describes the interface of SUIF compiler infrastructure. Chapter 4 discusses the implementation and execution of SDF compiler based on SUIF. The results of running benchmark programs and comparisons are described in Chapter 5. In Chapter 6, conclusions are presented and also a brief discuss of the future work.
2 SDF MACHINE MODEL

2.1 Decoupling Memory Access from Execution Pipeline

Memory latency is the bottleneck preventing modern computer architectures from gaining high performance. The use of larger cache can alleviate the memory latency problem, but it increases the cache hit access time. In addition, larger cache consumes more of the expensive silicon area on a chip. One way to eliminate these limitations is to use a decoupled computer architecture which separates operand access from execution and overlaps with execution. The separate instruction streams in a decoupled architecture communicate via queues [8]. In a traditional decoupled architecture, deadlocks can occur if both queues are full and both processors are blocked by the full queues, or if both queues are empty and both processors are blocked by the empty queues. However, SDF architecture can overcome these problems by using non-blocking threads.

The decoupled ideas were recently used in a multithreaded architecture known as Rhamma [9]. Rhamma uses conventional control-flow programming paradigm and blocking threads. This leads to more context switches in the Rhamma architecture. SDF architecture employs a non-blocking model which frees programmers from considering the synchronization management between threads. This also eliminates the context switches caused by blocking threads.

2.2 Dataflow Model and Architectures

A dataflow model processor executes an instruction at the earliest possible time upon the availability of both the input operands and a functional unit on which to execute that instruction [12]. To execute instructions in dataflow model, a compiler is required to provide the information about the dependencies between instructions. In the dataflow
model, an instruction is a successor of another instruction if it uses as one of its operands
the result of that other instruction. When an instruction completes, it will deliver its result
to each of its successor instructions. The property of data triggering the fetching and
execution of an instruction is a better strategy than the instruction-level speculative
execution since every instruction executed is useful and the processor does not waste any
cycle due to error conditions raised by the speculative operations. For the reasons above,
we did not implement instruction-level speculation in SDF architecture, but implemented
thread execution in dataflow model.

The data flow model is one of the basses for exploiting massive parallelisms
presented in many computer applications [10]. Although the dataflow model and
architecture have been studied for more than two decades, few dataflow processors have
been made since the actual implemention model has failed to deliver the promised
performance. Nevertheless, several features of dataflow computational model have been
found in the implementation of modern processors and compilers, such as static single
assignment, register renaming, reservation stations, and I-structure. Most modern
processors use complex hardware techniques to detect data and control dependences,
which actually implements an idealized dataflow model [11]. The pure dataflow
architecture has the following limitations: 1) fine-grained (instruction level)
multithreading; 2) difficulty in exploiting memory hierarchies and registers; 3)
asynchronous triggering of instructions. Many researchers have addressed the first two
limitations. Our SDF architecture specifically addresses the third limitation [1]. The
instructions within a SDF thread still retain the functional properties of the dataflow
model, while they are scheduled sequentially. SDF architecture eliminates Write-After-Write (WAW) and Write-After-Read (WAR) dependencies.

2.3 The Scheduled Dataflow Processor

In a SDF thread execution, a program is partitioned into non-blocking threads. Data are preloaded into an enabled thread’s context prior to its scheduling on the execution pipeline. After the thread is completed, it will post-store the results on which its successor threads depend. A thread is a successor of another thread if the thread uses the post-stored result of the other as one of inputs of execution. To synchronize the execution of a thread, a SDF processor uses two separate processing elements which communicate via queues. The two processing elements are called: the synchronization processor (SP) which performs memory accesses, and the execution processor (EP) which implements the operations of integer or floating point. All the operands executed on the EP are prepared by a SP and a SP accesses memory and moves data between memory and registers.

A SDF thread consists of three parts: preload, execute, and poststore. Preload and poststore parts are executed on the SP, while execute part is executed on the EP. A thread in memory is represented by a frame, which is a chunk of memory. A thread has a synchronization count, which stands for the number of inputs needed by that thread. When the number of inputs for a thread reaches its Synchronization Count, the thread becomes executable. A special instruction FALLOC operates the creation of a thread. The creation of a thread needs two parameters: thread name and synchronization count. After the creation of a thread, a pointer to the frame allocated for that thread is stored in a register. When a thread completes the execution, the resources used by that thread will be
released. The thread may post-store data to its successor threads. Once a thread is scheduled to a functional unit, it will execute until completion (that is non-blocking model). Hence, unlike other multithreading architectures, in the SDF architecture, there are no preemptive context switches, which have negative effects on performance.

A thread is uniquely identified by its continuation. A continuation in the SDF architecture is a simple four-value tuple \(<\text{FP}, \text{IP}, \text{RS}, \text{SC}>\), where FP is a Frame Pointer (where thread input values are stored), IP is an Instruction Pointer (which points to the thread code), RS is a Register Set (a dynamically allocated register set), and SC is a \textit{Synchronization Count}. A SDF thread has an associated continuation that is controlled by a Scheduler Unit (SU). At a given time a thread continuation can be one of the following, where “--“ means that the value is either undefined or unnecessary:

- **Waiting Continuation (WTC) or \(<\text{FP}, \text{IP}, --, \text{SC}>\)** - thread is waiting for inputs
- **Pre-Load Continuation (PLC) or \(<\text{FP}, \text{IP}, \text{RS}, -->\)** - in preload status
- **Enabled Continuation (EXC) or \(<--, \text{IP}, \text{RS}, -->\)** - in execute status
- **Post-Store Continuation (PSC) or \(<--, \text{IP}, \text{RS}, -->\)** - in poststore status

Thus, at a given time a thread can be in one of four possible states: WTC, PLC, EXC, or PSC (Figure 2-1) based on its continuation [1]. A SU handles the management of continuations and processing resources. The creation of a thread means that the Scheduler Unit allocates a \textit{Frame} to that thread and that thread will be in status of WTC. After completing the phase of creation, a thread’s continuation moves from "pre-load" (or PLC) status at SP, to "execute" (or EXC) status at EP and finishes in "post-store" (PSC) status again at SP.
Figure 2-1  Thread continuation transitions handled by the scheduling unit (SU). (Used with permission J. Arul, 2001, unpublished dissertation, the University of Alabama in Huntsville)
3 SUIF INTERFACE

The SUIF (Stanford University Intermediate Format) is a retargetable infrastructure for advanced parallelizing compiler research [13]. The SUIF is easy to use and build upon. SDF architecture depends on a compiler to analyze the dependencies between instructions, to identify the threads, and to explore the thread level parallelism at the compile time. The SDF compiler derives a sequence of threads from a sequential program and these threads are parallelized to execute on a SDF architecture. Compared with other available retargetable compilers, the SUIF libraries include a number of generic data structure classes and these classes are much easier to be modified and extended, which makes SUIF an ideal platform for developing SDF compiler.

3.1 Overview of the SUIF1.0 Hierarchy

All major SUIF1.0 data structures are derived from the `suif_object` class. SUIF’s major classes include `file_set_enry`, `tree_node`, `sym_node`, `type_node`, `instruction`, and `var_def`. The class of `file_set_enry` contains a global symbol table and a list of the files being compiled. Different kinds of AST (Abstract Syntax Tree) nodes are derived from the `tree_node` class. These nodes include `tree_instr`, `tree_if`, `tree_for`, `tree_block`, and `tree_proc`. The class of `sym_node` is the base class for all SUIF symbols. There are three kinds of symbols in SUIF: variable symbol, label symbol, and procedure symbol. The class of `type_node` is the base class of SUIF types. Each `instruction` node holds a SUIF instruction. SUIF supports both expression trees and flat lists of instructions. The class of `var_def` not only identifies the place where the variable is defined, but also specifies the alignment restriction for the variable’s storage [14].
Figure 3-1 Class Hierarchy of SUIF1.0 Data Structures.

The global symbol information is kept at the top level of SUIF intermediate representation. Information about each input and output file is kept at the second level. The Abstract Syntax Tree (AST) which is well-suited for dependence analysis and loop transformation is at the third level. At the bottom level, there are the expression trees.

3.2 Intermediate format of SUIF

The SUIF intermediate format is a mixed-level program representation. Besides the conventional low-level operations such as expression trees, it includes three high-level constructs: loops, conditional statements, and array accesses [13]. The intermediate format is language-independent and provides the necessary information for implementing parallelization techniques by compilers.
The intermediate representation of programs in SUIF is a crucial element to construct a compiler [13]. If the intermediate representation is in too low level, it is difficult to use such program representation to extract parallelisms of a program. For example, analyzing data dependency is difficult if the array accesses are expanded into arithmetic address calculations. At another extreme, if the representation is too high level, it will become language-dependent. The following figure illustrates what the intermediate format of SUIF looks like.

```c
int main(){
    int i;
    int array[100];
    for (i=0; i < 100; i++)
        array[i] = array[i] + 1;
}
```

C source code | intermediate format of SUIF
--- | ---

Figure 3-2 Intermediate representation of SUIF.

3.3 Automatic Parallelization Supported by the Infrastructure of SUIF

SUIF consists of a small kernel and a toolkit. The small kernel defines the intermediate format and the toolkit provides various compilation analysis and optimization techniques built using the kernel. All program information necessary for implementing analysis, parallelization, and optimization of program compilation is available with the SUIF [13].
At present, most parallel programs have to be explicitly managed by the programmers, for example using Pthread and MPI libraries. Developing an efficient parallel program requires a highly knowledgeable programmer. Moreover, the explicit parallel programming is tedious and error prone [15]. An automatically parallelizing compiler should automatically locate parallel computations in a sequential program. It frees programmers from the difficult tasks of explicitly managing parallelism of a program. The efficiency of programs automatically parallelized is another important issue for an automatically parallelizing compiler. In fact, it is not uncommon that a parallel program runs slower than the serial counterpart due to the overhead of synchronization [17]. For some programs, the smaller granularity of thread-level parallelism, the more overhead of managing thread synchronization. That means increasing the granularity of thread-level parallelism of programs can reduce the frequency of synchronization and minimize the overhead of managing parallelism between the parallel threads. In order to generate efficient parallel threads from a program, the SDF parallelizing compiler has to locate coarse-grain parallelism and reduce the number of times of pre-loads and post-stores among threads. SUIF provides powerful support of parallelization analysis.
techniques to implement SDF compiler, including scalar analysis, symbolic analysis, array analysis and interprocedural analysis as well as some optimization techniques [17].
There are three limitations in superscalar and SMT architectures due to their complex hardware: 1) the processor core used in superscalar and SMT architectures consumes a significant amount of silicon area; 2) these two architectures require longer pipeline due to the implementation of branch prediction and instruction speculative execution; 3) their cores are complicated and composed of many closely interconnected components [21]. There are no such problems in a single-chip multiprocessor since each core of a single-chip multiprocessor keeps the hardware structure simple. However, the challenge is not whether we can build a cost-effective single-chip multiprocessor, but rather whether we can exploit the parallelisms of an application very well. The tasks of parallelizing a program do not have to be imposed on programmers since writing a parallel version of the program which both behaves correctly and achieves good performance can be very difficult. A parallelizing compiler should automatically convert a program into its parallel version that retains correct semantics of the program and has efficient performance [22]. Automatic parallelizing is not a new idea, and researchers have been working on this topic for many years. In this chapter, our latest new research about automatic parallelization based on SDF architecture will be described. We developed a SDF automatic parallelizing compiler based on the SUIF infrastructure that can convert a conventional program into SDF threads that can be executed concurrently. SDF compiler is composed of two parts: a front-end that partitions the sequential program into threads represented in the SUIF intermediate format; a back-end that

* Work reported here is done jointly with Dr. Litong Song and Wentong Li.
generates SDF threads and instructions from the intermediate representations. Figure 4-1 shows the infrastructure of SDF parallelizing compiler.

4.1 Front End

4.1.1 Creating SDF Threads

In general, there are two ways to partition a program into threads: control-driven and data-driven [23]. Control-driven partitions a program into threads along the control flow boundaries while data-driven extracts threads from a program along the data flow boundaries. The challenge of control-driven multithreading is to find the division points that minimize the cost of inter-thread synchronization management. In data-driven multithreading, instructions from one or several computation parts of a program are packed into threads implicitly ordered by the data flow relationship between threads. Either in control-driven multithreading or in data-driven multithreading, threads extracted from programs are expected to execute across the available processing elements on a SDF single-
chip multiprocessor architecture. High performance is obtained from the overlapped execution of data-independent threads.

A SDF thread is composed of a pre-load list that contains the inputs needed by that thread to execute, a body part that includes most instructions of one thread, and a post-store list that stores the outputs to the successor threads having data dependent relationships with the current thread. SDF architecture supports to combine control-driven and data-driven together to partition a program into threads.

In our first implementation version of SDF compiler, we adopt the control-driven method to partition a program into threads. A program is represented using a control-flow graph (CFG). In a CFG, each node is a basic block and a directed edge from one node to another indicates the possibility of a control flow from one basic block to another. A basic block is a sequence of consecutive intermediate quadruple statements whereby a flow of control always enters at the beginning and leaves at the end without halting or possibility of branching except at the end. We use one basic block as a thread partition unit. In order to identify a thread, we implement liveness analysis on scalar variables and array variables, and inter-procedure analysis to recognize data dependencies between threads. Figure 4-2 illustrates the code of a SDF thread.
In order to reduce the frequency of synchronization and minimize the overhead of managing parallelism between threads, an extended basic block can be used as a thread partition unit to divide programs. An extended basic block is defined as a sequence of basic block $B_1 \ldots B_k$, such that for $1 \leq i < k$, $B_i$ is the only predecessor of $B_{i+1}$ and $B_i$ does not have any predecessor [24]. Some compiler optimization techniques, such as instruction scheduling, common expression elimination, and partial redundancy elimination, are used along with extended basic block techniques in order to obtain coarse-grained threads [25].

In the future, we will implement thread partition in different scopes to minimize the cost of managing threads.

4.1.2 Thread-level Speculative Execution

Speculative execution in instruction level is crucial for performance enhancement of IPC (Instruction per Cycle) in a traditional architecture extracting
ILP. For example, Intel Itanium Processor has adopted this technique [26].

Speculative execution at instruction level is implemented by performing loads as early as possible and inserting recovery code at the checkpoint. Compiler-controlled speculative execution includes control speculation and data speculation. Control speculation refers to breaking control dependences that occur between branches and other operations while data speculation refers to breaking data dependences between memory access operations [27].

Speculative execution in the thread level is analogous to the instruction-level speculation. Thread-level data speculative execution has been received much attention from researchers in the Stanford Hydra, Wisconsin Multiscalar, and Carnegie Mellon STAMPede projects [18, 19, 23]. These architectures all perform data speculative execution and depend on detection and recovery from violations of the independence assumptions.

SDF architecture and program diagram support the thread-level control speculative execution. The body and test of a loop are executed simultaneously and the test is determined later. The mechanism of recovery is very simple if the assumption failed. It will release the memory allocated for misspeculated threads by using a special instruction FREE. The mechanism is software-centric and does not need hardware to detect violations, which are required in other architectures. Figure 4-3 illustrates the SDF thread speculative execution in a loop.
4.1.3 Loop Transformations

Program loops provide a great opportunity for exploiting thread-level parallelism of a program. SDF compiler splits a loop up so that a number of iterations of the loop can be allocated to different threads, which can be executed on different processing elements of SDF architecture if the different iterations do not have loop-carried dependence [3]. The goal of such parallelization efforts is to maximize the resource utilization and to facilitate higher parallelism at thread level. Based on the front-end analysis of SDF compiler, we know that parallelizing only the inner loops is not enough for utilizing resources due to two reasons: 1) inner loops may not make up a significant part of the sequential program; 2) the cost of synchronization management will overwhelm the benefits of the parallel execution since the threads are fine-grained [28]. For example, if an inner loop only has one instruction and different iterations of the loop have loop-carried dependence, although we partition each iteration of this loop into one
thread, we will not get high performance due to the cost of thread synchronization. Hence, in order to reduce the cost of thread synchronization management, we parallelize not only the inner loops, but also the outer loops by partitioning iterations without loop-carried dependence into a thread.

The diagram below illustrates conceptually how SDF compiler might parallelize a loop into five threads (or k, a constant number) that can be executed simultaneously on the available processing elements of SDF architecture.

In order to parallelize loops, optimization techniques have to recognize opportunities for privatization and reduction of scalar and array variables, which have iteration-related dependences when we implement loop transformations [28]. Privatization means that each thread keeps a private copy of a scalar or an array variable. Reduction refers to computations of a sum, product, or maximum over scalar or array variables. SDF compiler supports the implementation of these optimization techniques.
We use dependence analysis on scalar variables to implement *privatization* and *reduction* techniques in the SDF compiler. In the example below, there is no dependence for the variable *total* between different iterations in the outer loop, so we implement *privatization* technique on threads partitioned from the outer loop. However, there exists loop-carried dependence for the variable *total* between different iterations in the inner loop, so we implement *reduction* technique on threads divided from the inner loop. Therefore, we can remove the dependence between different iterations and parallelize loops by implementing *privatization* and *reduction* techniques on the outer loop and the inner loop, respectively.

```c
for(i=0; i<n; i++)
    for(j=0, total=0; j<m; j++)
        total +=a[i][j]
        b[i]= c[i]*total

(a) before optimization
```

```c
#implement privatization on total
for(i=0; i<n; i++)
    for(j=0, total=0; j<m; j++)
        total +=a[i][j];
        b[i] = c[i]*total

(b) after optimization
```

Figure 4-5 Privatization and Reduction transformation.

4.2 Code Generation

4.2.1 Thread Generation

After partitioning a program into SDF threads, the program is represented using thread data-flow graph (TDG). In a TDG, each node is a thread and a directed edge between two nodes indicating that the *successor* thread has data dependence on its *predecessor* thread. Each node except root node may have multiple *predecessors* and multiple *successors*. The root node represents the main
function of a program, so it does not have an entry edge. The depth of a node refers to the longest path from the root node to that node. Any two threads in which there is no data dependence between each other can be executed simultaneously on different processor elements if hardware resources are available.

Generating a thread for SDF is composed of three parts: pre-load list generation whereby a thread will get the inputs from its predecessor threads; computation part generation whereby a thread completes the functionality of calculation; post-store list generation whereby a thread will store the outputs needed by its successor threads.

4.2.2 Cascaded If-statements Optimization

A SUIF tree_if contains three tree node lists: the header, the then_part, and the else_part. Each header contains the code to test the “if” condition and a branch to the jumpto label, implicitly located at the beginning of the else_part. SUIF breaks cascaded If-statements into low-level branches and jumps with explicit labels [14]. For complex test computations, such as an evaluation expression including && and || operators, SUIF views each expression between these special operators as a tree_if node. If-statements in SDF are implemented with FORK statements. Therefore, cascaded if-statements in SUIF should be converted into formats that SDF architecture can recognize (see Figure 4-6 for an example).
if ((x<y) && (x > 0)){
    y = x;
} else{
    y = 0;
}

(a) if source code
(b) SUIF tree_if node
(c) SDF if-test code

Figure 4-6  Simplified illustration of cascaded if-statement.

4.2.3 Constant Propagation

At the back end, we do not have the whole picture about a program.

Hence, we implement local constant propagation within a thread at the thread
generation. Within a thread, assuming that a constant variable \( x \) defined in
statement \( S_1 \) and value of \( x \) is used in the statement \( S_k \), we can replace variable \( x \)
in the statement \( S_k \) with its constant value. First, we convert the tree node format
of SUIF instruction into the quadruple format with a destination operand and two
source operands. Then we implement constant propagation algorithm on the new
instruction list.
In this chapter we will compare a number of different architectures such as SDF (scheduled data-flow), superscalar out-of-order execution, VLIW (very long instruction word), and SMT (simultaneously multithreading) using a set of benchmarks, i.e. matrix multiply, zoom from the core part of a picture zooming program, jpeg from the MediaBench⁴ test suites, and autocorrelation from the EEMBC (Embedded Microprocessor Benchmark Consortium) test suites. The effect of thread parallelizing level on the SDF execution behavior has also been explored by changing the parallelism parameters in SDF parallelizing compiler.

5.1 Methodology

Each of the test benchmarks was compiled and run on a number of separate simulated machines. We evaluated SDF architecture by running codes generated by the SDF compiler on the SDF architecture simulator [1]. SimpleScalar [29] tool set was used to compile the test programs and simulates the out-of-order execution of superscalar architecture, which allows us to configure different simulation environments for different experimental purposes. Trimaran [30] simulator was used to simulate VLIW machines and it also provided the parameters to be configured for different simulation environments. Ssmt-1.0[20] derived from SimpleScalar tool set simulated the execution of SMT architecture.

We investigated the execution performance gained by changing the number of SPs and EPs in SDF and compared the execution performance of SDF with superscalar and VLIW processors containing the same number of functional units. Since there is not

---

⁴ MediaBench test suites available at [http://cares.icsl.ucla.edu/MediaBench](http://cares.icsl.ucla.edu/MediaBench).
a multithreading compiler for SMT, SMT simulates multithreading by executing multiple programs simultaneously. Therefore, we compared SDF with SMT architecture with multiple contexts running on these architectures and each context contains an application as one thread.

Matrix multiplication can be written to exploit both thread level and instruction level parallelism. Zoom consists of three nested loops and a substantial amount of instruction level parallelism in the middle loop [1]. There are four nested loops in the jpeg program whereby we explore the thread level parallelism at two outer loops and each thread has longer run-lengths. Autocorrelation is composed of loops whereby we can explore the thread parallelism and each thread has shorter run-lengths. Loop unrolling can increase the instruction level parallelism and avoid the execution times of conditional test instruction at loop boundary [3]. We unrolled the instructions of the array computation operations inside of a loop. To extract more thread-level parallelism, loop transformation techniques previously described are implemented. A loop can be partitioned into k threads and k can be a varied number. Increasing the value of k will put more pressure on register allocation. Moreover, an increase of k will dramatically increase the code size of one thread. In this work, k is set to 4.

In Figure 5-1, 5-2, 5-3, 5-4, and Figure 5-5, xI represents x integer functional units and xF stands for x floating functional units. xE represents xEPs and xS stands for xSP in SDF architecture.

5.2 Experimental Results for superscalar Architecture.

In SDF experiments, we spawned a fixed number of 4 threads of each loop at one time. We have investigated the performance of SDF using multiple SPs and EPs and
compared the performance with superscalar architecture using multiple Integer and Floating-Point units. The number of functional units in superscalar architecture is equal to the number of EPs and SPs in SDF [1]. Table 5-1 shows the data for these series of experiments.

Table 5-1 SDF vs. superscalar using varying number of functional units

<table>
<thead>
<tr>
<th></th>
<th>SS</th>
<th>SDF</th>
<th>SS</th>
<th>SDF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(cycles)</td>
<td>(cycles)</td>
<td>(cycles)</td>
<td>(cycles)</td>
</tr>
<tr>
<td>1INT ALU</td>
<td></td>
<td>1SP</td>
<td>2INT ALU</td>
<td>2SP</td>
</tr>
<tr>
<td>1FP ALU</td>
<td></td>
<td>1EP</td>
<td>2FP ALU</td>
<td>2EP</td>
</tr>
<tr>
<td>Matrix</td>
<td>9,950,854</td>
<td>9,097,391</td>
<td>4,990,506</td>
<td>4,690,002</td>
</tr>
<tr>
<td>Zoom</td>
<td>13,468,532</td>
<td>14,014,135</td>
<td>8,237,862</td>
<td>7,752,739</td>
</tr>
<tr>
<td>Jpeg</td>
<td>259,571</td>
<td>246,122</td>
<td>149,801</td>
<td>134,928</td>
</tr>
<tr>
<td>Autocorrelation</td>
<td>20,723</td>
<td>27,597</td>
<td>11,899</td>
<td>15,058</td>
</tr>
<tr>
<td>Matrix</td>
<td>3,540,437</td>
<td>3,017,624</td>
<td>3,537,223</td>
<td>3,016,401</td>
</tr>
<tr>
<td>Zoom</td>
<td>6,351,642</td>
<td>5,245,830</td>
<td>6,236,702</td>
<td>4,848,513</td>
</tr>
<tr>
<td>Jpeg</td>
<td>139,013</td>
<td>100,847</td>
<td>139,008</td>
<td>98,401</td>
</tr>
<tr>
<td>Autocorrelation</td>
<td>9,605</td>
<td>9,986</td>
<td>9,302</td>
<td>9,012</td>
</tr>
</tbody>
</table>

SS stands for “superscalar”

Figure 5-1 shows that SDF outperforms superscalar out-of-order execution in all cases of matrix multiplication, zoom, and jpeg. The results indicate that exploring the thread level parallelism has significant effect on achieving high performance. Although autocorrelation has thread level parallelism, SDF cannot explore thread level parallelism very well with limited number of functional units, but with increasing the number of SPs and EPs, SDF outperforms superscalar. This experimental result testified again that SDF architecture is more scalable than superscalar architecture.
Figure 5-1  Performance Comparison of SDF and superscalar architecture.

Figure 5-2 shows that SDF is more scalable than superscalar in IPC. The more thread level parallelism of applications, the more scalable is the performance in SDF. In SDF, hardware resources are added to extract more thread-level parallelism of programs. However, in superscalar more hardware resources are added to enhance the ability to exploit instruction-level parallelism. The experimental results demonstrated that exploiting TLP has more significant effect on achieving high performance than extracting ILP for computer architectures.
5.3 Experimental Results for VLIW Architecture.

In the Very Long Instruction Word (VLIW) or EPIC (explicitly Parallel Instruction Computer) architectures, each instruction packet contains multiple, independent instructions called operations. Each of these operations is executed on available functional units. Loop unrolling can produce more independent instructions for VLIW architecture. Trimaran simulator is use to evaluated the performance of VLIW architecture based on HPL-PD architecture. We investigated the performance of SDF and VLIW architecture by varying the number of functional units. Table 5-2 contains the data of this experiment. Figure 5-3 and Figure 5-4 graphically depicts that SDF has significant performance and scalability compared to VLIW. Figure 5-3 shows execution performance in terms of total execution cycles. Figure 5-4 shows IPC comparisons.
Table 5-2 SDF vs. VLIW using varying number of functional units

<table>
<thead>
<tr>
<th></th>
<th>VLIW (cycles)</th>
<th>SDF (cycles)</th>
<th>VLIW (cycles)</th>
<th>SDF (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1INT ALU</td>
<td>1SP</td>
<td>2INT ALU</td>
<td>2SP</td>
</tr>
<tr>
<td>Matrix</td>
<td>9,377,547</td>
<td>9,097,391</td>
<td>9,351,227</td>
<td>4,690,002</td>
</tr>
<tr>
<td>Zoom</td>
<td>13,174,030</td>
<td>14,014,135</td>
<td>11,486,478</td>
<td>7,752,739</td>
</tr>
<tr>
<td>Jpeg</td>
<td>217,038</td>
<td>246,122</td>
<td>200,046</td>
<td>134,928</td>
</tr>
<tr>
<td>Autocorrelation</td>
<td>10,370</td>
<td>27,597</td>
<td>11,151</td>
<td>15,058</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>VLIW (cycles)</th>
<th>SDF (cycles)</th>
<th>VLIW (cycles)</th>
<th>SDF (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4INT ALU</td>
<td>4SP</td>
<td>6INT ALU</td>
<td>6SP</td>
</tr>
<tr>
<td>Matrix</td>
<td>9,309,227</td>
<td>3,017,624</td>
<td>9,289,179</td>
<td>3,016,401</td>
</tr>
<tr>
<td>Zoom</td>
<td>9,340,174</td>
<td>5,245,830</td>
<td>9,274,638</td>
<td>4,848,513</td>
</tr>
<tr>
<td>Jpeg</td>
<td>162,408</td>
<td>100,847</td>
<td>159,298</td>
<td>98,401</td>
</tr>
<tr>
<td>Autocorrelation</td>
<td>12,030</td>
<td>9,986</td>
<td>12,902</td>
<td>9,012</td>
</tr>
</tbody>
</table>

Figure 5-3 Performance Comparison of SDF and VLIW architecture.
5.4 Experimental Results for SMT Architecture.

Simultaneous Multithreading architecture (SMT) extends from superscalar architecture. It replicates the functional units of a superscalar architecture and requires a powerful instruction fetching hardware unit to dynamically patch independent instruction from multiple threads on the available function units [4]. In SMT architecture, multiple threads are independent instruction streams from either the same or different programs. Ssmt-1.0 [20] simulator simulates SMT architecture by executing different programs concurrently to explore thread level parallelism. For each experiment, there are three combinations: 2 programs in parallel with 2 copies of functional units, 4 programs in parallel with 4 copies of functional units, and 6 programs in parallel with 6 copies of functional units. We use IPC (instruction per cycle) to evaluate the scalability of the architecture performance. In general, IPC is a measure of parallelism that can be misleading, since the techniques of branch prediction and instruction speculative execution make IPC not an accurate measure of parallelism. However, these techniques
extracting aggressively ILP are not implemented in SDF architecture. Therefore, using IPC to compare SDF with SMT and superscalar can not be misleading. In SDF, all executed instructions are useful instruction. Table 5-3 and Figure 5-5 show that with increasing number of the hardware resources, SDF is more scalable than SMT architecture.

Table 5-3 SDF vs. SMT by executing multiple programs in parallel.

<table>
<thead>
<tr>
<th></th>
<th>SMT (IPC) 2 contexts</th>
<th>SDF (IPC) 2 copies</th>
<th>SMT (IPC) 4 contexts</th>
<th>SDF (IPC) 4 copies</th>
<th>SMT (IPC) 6 contexts</th>
<th>SDF (IPC) 6 copies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>2.0246</td>
<td>1.9840</td>
<td>3.6285</td>
<td>3.5899</td>
<td>3.7539</td>
<td>5.1510</td>
</tr>
<tr>
<td>Zoom</td>
<td>1.6588</td>
<td>1.7967</td>
<td>2.1541</td>
<td>3.6164</td>
<td>2.1744</td>
<td>5.1263</td>
</tr>
<tr>
<td>Jpeg</td>
<td>1.9959</td>
<td>1.9973</td>
<td>3.3281</td>
<td>3.2996</td>
<td>3.4068</td>
<td>4.4944</td>
</tr>
<tr>
<td>Autocorrelation</td>
<td>1.9673</td>
<td>1.7180</td>
<td>3.2935</td>
<td>3.1370</td>
<td>3.3493</td>
<td>4.6544</td>
</tr>
</tbody>
</table>

5.5 Experiments for Effect of Parallelizing Level on Performance.
We also investigated the scalability of SDF by exploring different parallelizing levels. We tested four programs, matrix multiplication, zoom, jpeg, and autocorrelation. For each test program, we generated three versions using SDF compiler by spawning different number of threads: 1, 2, and 4 respectively, which is represented by different grays in Figure 5-6, 5-7, 5-8, 5-9. We executed these three versions of each program on different SDF machines with different number of processing elements.

These results emphasized the significant effect of automatic parallelization on achieving high performance in SDF architecture. As indicated by the data, the loop transformation shows higher reductions in execution cycles even in the case of 1 SP and 1 EP. Loop transformation techniques explored more thread level parallelism, which made more overlapping of SP and EP execution than without loop transformation implementation. If increasing the parallelization level, we can get significant improvement on execution cycles. At the same time, increasing the number of processing elements also benefits the performance improvement. With 6 SPs and 6 EPs hardware resources when the parallelization level is increased by one time, the execution cycles almost can be reduced by fifty percent. On the other hand, we noticed that for matrix multiplication, although we increase the number of processing elements from 4 to 6, the performance does not show a corresponding improvement. The performance may improve if the compiler can generate more independent threads. However, the number of registers available per thread context limits the number of threads that can be forked in parallel.

Based on the hardware resources and the application, we can use the SDF compiler to configure dynamically the thread level parallelism execution on SDF
architecture. In our ongoing research, we will explore these issues related to configurable compilation on SDF architecture.

![Matrix Multiplication Performance with different parallelizing level and processing elements](image1)

**Figure 5-6** Evaluating effect of different parallelizing level of matrix multiplication.

![zoom performance with different parallelizing level and processing elements](image2)

**Figure 5-7** Evaluating effect of different parallelizing level of zoom.
Figure 5-8  Evaluating effect of different parallelizing level of jpeg.

Figure 5-9  Evaluating effect of different parallelizing level of autocorrelation.
6 CONCLUSIONS

SDF architecture has many desirable features of the next generation architecture: multiple processors on a single chip, less hardware complexity, and higher scalability. SDF computing model provides a powerful support for the state-of-art architecture. Most cycles in the SDF thread execution perform useful computation of the program. In our work, we showed that with the support of compiler optimizations, SDF is more scalable than other modern architectures such as superscalar, VLIW, and SMT. We also reported loop transformation techniques at compile time that can be used to increase both instruction level parallelism (ILP) and thread level parallelism (TLP) of a program while eliminating significant numbers of conditional instructions. By implementing automatic parallelization and loop transformation techniques in the SDF compiler, we achieved significant improvements of performance in terms of execution cycles and in scalability. In our ongoing research we will explore additional computer architectural innovations and compiler techniques together to achieve higher performance.
REFERENCES


and Speculative Execution in the IMPACT EPIC Architecture”. *Proceedings of

[28] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. S. Lam,
“Interprocedural Parallelization Analysis: A Case Study.” *Proceedings of the 8th
International Workshop on Languages and Compilers for Parallel Computing
(LCPC95)*, August 1995.


Version 1.1.” Compiler and Architecture Research HPL-93-80 (R.1), February
2000.