# Loop Scheduling for Multithreaded Processors

Georgios Dimitriou
*Dept. of Computer Engineering*
*University of Thessaly, Volos, Greece*
*dimitriu@uth.gr*

Constantine Polychronopoulos
*Dept. of Electrical and Computer Engineering*
*University of Illinois at Urbana-Champaign*
*cdp@csrd.uiuc.edu*

## Abstract

*The presence of multiple active threads on the same processor can mask latency by rapid context switching, but it can adversely affect performance due to competition for shared datapath resources. In this paper we present Macro Software Pipelining (MSWP), a loop scheduling technique for multithreaded processors, which is based on the loop distribution transformation for loop pipelining. MSWP constructs loop schedules by partitioning the loop body into tasks and assigning each task to a thread that executes all iterations for that particular task. MSWP is applied top-down on a hierarchical program representation, and utilizes thread-level speculation for maximal exploitation of parallelism. We tested MSWP on a multithreaded architectural model, Coral 2000, using synthetic and SPEC benchmarks. We obtained speedups of up to 30% with respect to highly optimized superblock-based schedules on loops with unpredictable branches, and a speedup of up to 25% on perl, a highly sequential SPEC95 integer benchmark.*

## 1. Introduction

Modern processor archictures with multiple or wide issue pipelines provide the opportunity of exploiting loop parallelism at both the interation and the instruction level. However, loop scheduling techniques developed for such architectures may prove unsuitable for multithreaded processors, having adverse affect on their performance. For instance, the sharing of an L1 data cache does no longer readily justify a partitioning of the loop iteration space into disjoint sets of iterations; such a partitioning potentially increases cache misses. Similarly, since a thread context switch preserves the instruction schedule and the private register contents of the thread that is switched off the pipeline, a function call can be included in a software pipelined schedule; the function body can be assigned to a separate hardware thread.

In this paper we present a loop scheduling mechanism for multithreaded processors called *Macro Software Pipelining* (MSWP). MSWP utilizes the loop distribution transformation for loop pipelining [1]. An MSWP schedule of a loop is constructed by partitioning the loop body into well-balanced tasks in a way that minimizes data communication across tasks, and by assigning each task to a processor thread that executes all loop iterations for that task. For tasks of similar sizes and no backward dependencies in the loop body, an MSWP schedule appears like a software pipeline schedule with tasks replacing instructions. It also resembles a transposed doacross schedule, where each column contains consecutive instances of the same task, instead of different tasks from the same iteration. A backward dependence in the loop body can be handled by MSWP in one of two ways: (a) through thread synchronization that results in a delayed schedule, or (b) through thread-level speculation that defers the dependence enforcement until after the dependence sink starts a speculative execution.

The rest of this paper is organized as follows. In section 2 we will discuss loop scheduling at both the iteration and the instruction level, and its application on multithreaded architectures. We will focus our discussion on MSWP in section 3, where we will give details on the implementation of MSWP-based loop scheduling. In the following section we will briefly present our multithreaded architectural model. Section 5 will discuss the results of applying MSWP on a number of benchmark programs, obtained from a simulator of our model. We will close the paper with conclusions.

## 2. Loop scheduling and multithreading

### 2.1. Exploitation of parallelism in loops

Traditional loop scheduling for the exploitation of parallelism at the iteration level, is mostly performed by scheduling different loop iterations on separate processors [2]. If no loop-carried dependencies are detected on the target loop [3], a completely parallel schedule (namely, a *doall* loop) is produced. Otherwise, if there is adequate iteration overlap, synchronization operations are inserted in the loop body, resulting in a *doacross* loop. Alternatively, the loop can be partitioned across tasks of the loop body, instead of iterations, resulting in a *dopipe* loop. Loop distribution is used in this case for pipelining the execution of tasks and exploits functional in addition to iteration-level parallelism.

At the instruction level, the prevailing loop scheduling technique is *Software Pipelining* (SWP) [4]. Circular instruction reordering across multiple loop iterations is the key element in that mechanism. The *Minimum Initiation Interval* (MII) is the minimum number of cycles between consecutive iteration schedules in SWP, under certain code and architectural constraints. Instruction placement in the loop schedule through the *modulo constraint* allows the loop to be scheduled with the MII. In recent work on SWP, scheduling of control-intensive loops [5] is performed on *superblocks*, i.e. on paths with the highest execution likelihood [6]. Any branch out of the pipelined loop must lead into compensation code.

## 2.2. Multithreaded architectures

Multithreaded processors can support in hardware more than one active thread at the same time. Depending on the way of handling context switches among threads, the *blocked* multithreaded architectures switch context on demand only, whereas the *interleaved* multithreaded architectures switch context at each cycle.

Early interleaved multithreaded processors were relying on the presence of threads to successfully hide memory access latencies [7], and were thus suffering from poor single thread performance. The use of a superscalar processor as an underlying architecture boosted the performance of interleaved multithreading [8] and led to *Simultaneous Multithreading* (SMT) [9]. SMT further extends interleaved multithreading, so as to allow multiple threads to fetch instructions from the instruction cache at the same time.

Other designs combine multithreading with on-chip multiprocessors. The *Multiscalar* processor [10], for instance, connects a number of CPUs through a ring, allowing them to schedule communicating threads. CPUs that are assigned different iterations of the same loop can be executing speculatively. Similar approaches that utilize thread speculation are considered in [11,12,13,14].

Thread-level speculation has been the focus of multithreading research in the last decade. Such coarse-grained speculation is more complicated than instruction-level speculation and needs extensive compiler and hardware support [15,16,17].

## 2.3. Loop scheduling and multithreading

Although the above loop scheduling techniques deliver good performance on the processor architectures they were designed for, they are often unsuccessful in exploiting the capabilities of multithreaded processor architectures.

**2.3.1. Parallel loop scheduling and multithreading.** Extracting parallelism of inner loop nests provides a multithreaded processor with only bursts of threads, separated by possibly long intervals of single-threaded code. Unfortunately, loop parallelism is difficult to extract at the outer-loop level.

Doall schedules provide the processor with identical SIMD-style threads that will execute in a loose synchronized mode. That mode will in fact be close to lock-step mode in interleaved architectures. If a thread executes a long latency operation, e.g. a non-pipelined integer division, all other threads will soon run into the same operation, and will all have to compete and wait for the corresponding functional unit. In the case of longer latencies, as in a load with a secondary cache miss, even blocked multithreaded architectures could be affected.

Cache locality can be heavily affected by resource sharing in multithreaded processors. By partitioning the iteration space of a loop into disjoint sets of iterations and assigning each set on a separate thread, we force threads to have separate data sets. This could easily result in a large number of conflict and capacity misses.

**2.3.2. Instruction scheduling and multithreading.** Recent research on instruction scheduling through SWP can produce code that utilizes processor resources at a maximal level. Unfortunately, such techniques still suffer from failures at function calls and unpredictable branches. After any jump out of the SWP code, returning into it is extremely difficult, if at all possible. With multiple branches that are difficult to predict, the rate of jumps out of the SWP schedule can grow exponentially.

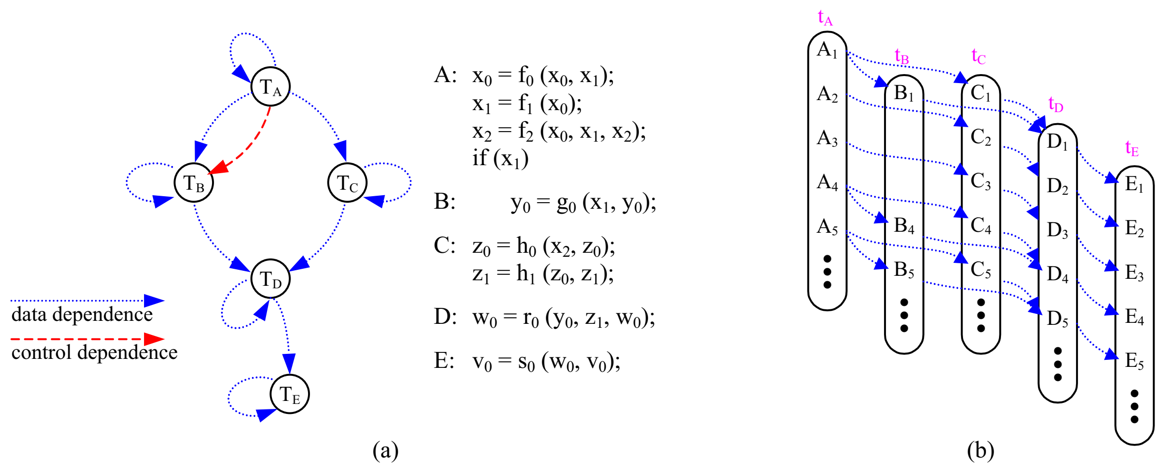
The occurrence of function calls and unpredictable branches is more frequent at higher levels of the program. Thus, application of SWP-based instruction scheduling is limited to inner – usually small – loops, not exploiting ILP at a scope that would result in an optimal performance for longer parts of a program.

Another common problem in SWP is register pressure. In order to obtain the necessary iteration overlap, SWP compilers apply loop unrolling before compacting the code. This, however, increases the requirements for register live ranges, since the code becomes longer, in effect limiting the unrolling factor.

## 3. MSWP overview

The characteristics of multithreaded processors suggest the use of loop distribution and the dopipe loop as the basis for MSWP. In this way, the dissimilarity of code in different threads results in a better exploitation of parallelism. Furthermore, any communication across threads forces them to work on the same or adjacent data sets, thus better exploiting spatial locality.

With dopipe loop scheduling, ILP can be further – and



$T_A$
$T_B$
$T_C$
$T_D$
$T_E$
data dependence
control dependence
(a)
$t_A$
$t_B$
$t_C$
$t_D$
$t_E$
$A_1$ $A_2$ $A_3$ $A_4$ $A_5$
$B_1$ $B_4$ $B_5$
$C_1$ $C_2$ $C_3$ $C_4$ $C_5$
$D_1$ $D_2$ $D_3$ $D_4$ $D_5$
$E_1$ $E_2$ $E_3$ $E_4$ $E_5$
(b)

A: $x_0 = f_0(x_0, x_1);$
$x_1 = f_1(x_0);$
$x_2 = f_2(x_0, x_1, x_2);$
if $(x_1)$
B: $y_0 = g_0(x_1, y_0);$
C: $z_0 = h_0(x_2, z_0);$
$z_1 = h_1(z_0, z_1);$
D: $w_0 = r_0(y_0, z_1, w_0);$
E: $v_0 = s_0(w_0, v_0);$

**Figure 1. MSWP scheduling example**

independently – explored through the application of SWP within each thread. By splitting the original loop into multiple loops, we reduce the size and the complexity of the loop body in each resulting loop. If the original loop could not be pipelined at the instruction level, it is now possible that several of the new loops will.

Additional exploitation of parallelism can be achieved through thread-level speculation, which allows iteration overlap to extend beyond the limits set by dependence cycles.

### 3.1. An illustrative example

An example for the application of MSWP on a loop is shown in Figure 1a. The loop body at a high-level language description is shown, together with the corresponding dependence graph, where both control and data dependence arcs are included. Instructions have been grouped into segments A through E, depicted on the five tasks $T_A$ through $T_E$ that are shown on the graph.

Each task is assigned into one of five threads $t_A$ through $t_E$, respectively, and the resulting schedule is shown in Figure 1b. Each thread, except $t_B$, executes all iterations for the corresponding task, communicating with other threads as necessary, in order to satisfy all dependencies. Thread $t_B$ executes only those loop iterations for $T_B$ that satisfy the control dependence between $T_A$ and $T_B$. That dependence is implicitly satisfied, whenever threads $t_A$ and $t_B$ communicate to satisfy the data dependence between the two tasks. Cyclic dependencies due to recurrences are automatically satisfied within each thread.

Iteration overlap, as a result of MSWP application, is clearly seen in the figure. Iterations across threads are scheduled asynchronously, e.g. $A_4$ can execute after $A_3$ is finished, even if thread $t_C$ is still working on $C_2$ or even $C_1$. Such an asynchronous scheduling of the loop tasks will be feasible, only if the underlying communication means can store intermediate values that are passed among the threads and produced in consecutive iterations.

### 3.2. The MSWP scheduling algorithm

The MSWP scheduling algorithm consists of the following four steps:

1. *Partition the code into threads.*
2. *Eliminate backward dependencies.*
3. *Insert code for communication across threads.*
4. *Apply instruction-level SWP.*

**3.2.1. Code partitioning.** Thread extraction is based on interprocedural dependence analysis and is performed at a top level of the code representation, i.e., by applying loop distribution on large outer loops. It moves recursively into inner loops, depending on relative task sizes, as calculated by a profiler. Code partitioning assumes a hierarchical representation of the input code through the Hierarchical Task Graph (HTG) [18], augmented with any necessary dependence and profile information. Major goal of code partitioning for MSWP is to extract the maximum potential iteration overlap in the execution of threads.

**3.2.2. Backward dependence elimination through thread-level speculation.** Speculation in the context of loop scheduling refers to speculatively scheduling an iteration, when it is dependent on one or more previous iterations. For data dependencies, each backward dependence is examined, to determine whether an early calculation of the value carried across the dependence is possible. The code for that calculation is then placed within the thread considered for speculation. Control dependencies that indicate control flow abnormalities, like early loop exits, are also checked for speculative elimination. Any compensation code necessary to handle mis-speculation is inserted at this step.

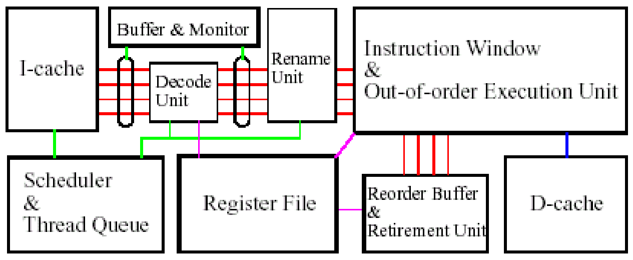


Figure 2. Coral 2000 architecture

**3.2.3. Communication code insertion.** Instructions are inserted, if existing code cannot satisfy communication requirements from data and control dependencies. This step is highly dependent on the particular communication means supported by the underlying architecture. In any case, instructions must be carefully inserted, in order to avoid communication deadlocks.

**3.2.4. Application of instruction-level SWP within threads.** SWP within a thread is meaningful, only if that thread executes all loop iterations. Otherwise, pipelining of consecutive iterations is likely to result in the execution of useless – and possibly harmful – code.

### 3.3. Parallelism exploitation in MSWP

A major advantage of MSWP over traditional loop scheduling techniques is that it can potentially exploit more parallelism. By distributing the body of a loop across threads, MSWP exploits functional-level parallelism first, leaving iteration-level parallelism to be exploited through asynchronous execution of loop iterations. Since doacross and doall loops do not exploit functional-level parallelism within the loop body, MSWP allows deeper parallelism exploitation. Instruction-level SWP, on the other hand, achieves such a goal through loop unrolling and code compaction, but at a smaller scale and with many limitations.

## 4. Coral 2000

Our processor model, namely *Coral 2000*, is a hybrid between blocked and interleaved multithreaded architectures, with a context switch mechanism based on the α-*Coral* concept [19]. A block diagram of Coral 2000 is given in Figure 2.

In the development of Coral 2000, we have considered support towards a successful implementation of MSWP. Its design goals include thus the following:

- A zero-overhead thread context switch mechanism that allows active threads to be switched without stalling the processor pipeline.
- A communication means for passing data across threads that allows threads to execute asynchronously, when they are working on a common loop, doing useful work instead of waiting for each other.
- The ability to switch threads at the fetch stage of the front-end of the processor, when conditional branches are encountered.
- A buffering mechanism that allows the front-end to mix instructions from more than one thread within the same stage, though instructions are still fetched from the same thread at each clock cycle.
- Support for thread-level speculation, in order to handle backward dependencies and control flow abnormalities.

Further details on Coral 2000 can be found in [20].

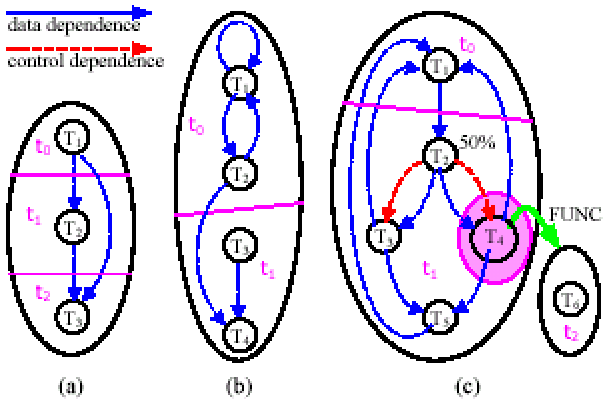


Figure 3. HTGs of the synthetic benchmarks

## 5. Experimental results

We tested and evaluated MSWP on a cycle-level simulator that was based on the *SuperDLX* simulator, a general-purpose simulator of a MIPS-like superscalar microprocessor [21]. We extended SuperDLX, to support most of the MIPS IV instruction set. We also added additional features, such as an L1 instruction cache with predecoding, a simple L1/L2 data cache hierarchy, register mapping tables, a multilatency integer multiply/ divide functional unit, and full store-to-load forwarding. Most importantly, we implemented all multithreading support, in accord with the design goals of Coral 2000.

### 5.1. Benchmarks

Goal of our testing was to prove that MSWP can successfully address the loop scheduling issues discussed above. To this end, we based our experiments mostly on synthetic benchmarks. Nevertheless, we also tested MSWP on *perl*, an integer SPEC95 benchmark.

All benchmarks are described through the HTGs of Figures 3 and 4. For simplicity, some nested conditional branches and loops are not shown. An outer loop is implied in all synthetic benchmarks. More specifically:

- *do*. The body of the outer loop of this program exhibits the dependencies shown in Figure 3a. Task $T_2$ contains

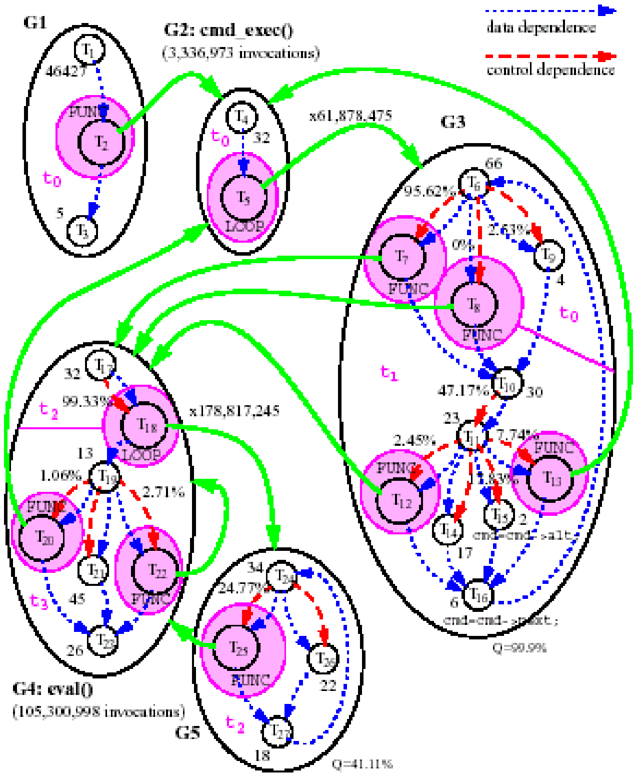


Figure 4. HTG of *perl*

long latency integer operations.

- *loop*. The body of the outer loop of this program exhibits the dependencies shown in Figure 3b.
- *func*. The body of the outer loop of this program exhibits the dependencies shown in Figure 3c. Task $T_4$ contains a function call, which is actually made in 50% of the loop iterations.
- *perl*. This program exhibits a highly sequential behavior, due to the backward dependencies shown in the HTG of Figure 4. The attached information is mostly a result of *gprof* on the execution of *perl* with the *ref* input data set. In particular, Q is a combined probability that the path carrying the respective backward dependence is followed. The code shown participates in speculation.

The partitioning of all programs into threads is also shown in the figures. The shaded compound tasks serve as thread interfaces.

## 5.2. Experiments

In order to evaluate MSWP, we compared execution of multithreaded to execution of single-threaded code. In each comparison, we used codes as similar as possible, with differences that resulted only from multithreading and the application of the MSWP algorithm. In order to obtain the best possible single-threaded codes, we applied superblock-based SWP on many of those codes. Thus, evaluation of MSWP was performed in the context of state-of-the-art instruction scheduling.

The basic simulator configuration we used, was that of a 3-way superscalar machine, with a reorder buffer of 40 entries and a store buffer of 36 entries. We assumed simple 32K L1 caches with one port, a hit time of 1 cycle and a miss time of 6 cycles. We ignored instruction misses, though. Finally, we used a single-level 2-bit dynamic branch prediction with a BTB of 512 entries.

For each program tested, we obtained results from four modes of simulation, depending on whether cache or dynamic branch prediction (dbp) was used.

**5.2.1. Synthetic benchmarks.** Figure 5 shows the simulation results for the three synthetic benchmarks. More specifically:

- Figure 5a depicts the execution times for *do*, using compiler-produced SWP codes, single-threaded (O3), 2-threaded with doall (2O3A), 4-threaded with doall (4O3A) and 3-treaded with MSWP dopipe (3O3P).
- Figure 5b depicts the execution times for *loop*, using superblock-based SWP codes, single-threaded without and with 2 times unrlolling (O2NU and O2U2), and 2-threaded MSWP code without, with 2 and with 4 times unrolling (2O2NU, 2O2U2 and 2O2U4).
- Figure 5c depicts the execution times for *func*, using superblock-based SWP codes, single-threaded without and with 2 times unrolling (O2 and O2U2), and 3-threaded MSWP code with 5 times unrolling (3O2U5C). In the last case, $t_2$ is conditionally executing only those iterations that actually perform a function call.

In all the above cases, using MSWP, we obtained speedups of up to 30% with respect to optimized single-threaded code. The unrolling factor used in each case was maximized, in order to obtain the optimal MII.

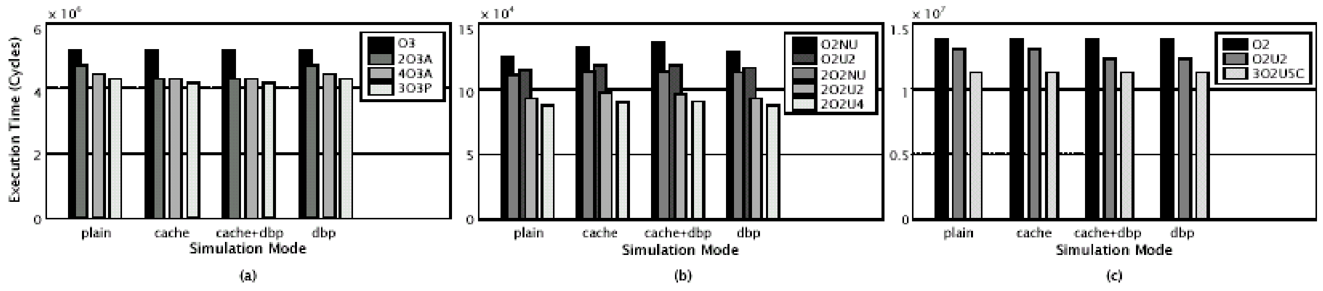


Figure 5. Execution times for the synthetic benchmarks

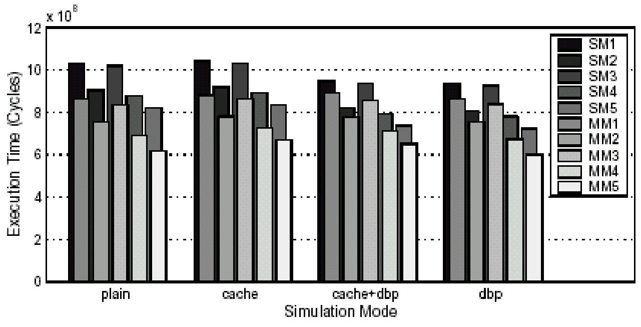


**Figure 6. Execution times for *perl***

**5.2.2. SPEC95 results.** In these experiments, we varied the machine configuration, from the basic (M1), to a 3-way superscalar with two (M2), a 4-way superscalar with one (M3), a 4-way superscalar with two (M4) and a 6-way superscalar with three memory ports (M5).

Figure 6 shows the execution times for *perl*, using a reduced *ref* input data set. We tested single-threaded (S) and 4-threaded (M) code, based on compiler-optimized code. In the application of MSWP we utilized thread-level speculation, mostly along the code included in Figure 4. The speedup we obtained was up to 25%.

## 6. Conclusions

In this paper we identify issues in traditional loop scheduling techniques, at both the iteration and the instruction level, that can be addressed quite successfully with multithreading through MSWP. With MSWP, we provide a loop scheduling technique that, unlike doall/ doacross scheduling, is more suitable to multithreaded architectures. MSWP exploits loop parallelism and additionally allows for a significant performance boost to instruction-level SWP for multithreaded processors. We propose an implementation of the MSWP algorithm and test it on synthetic and spec benchmark kernels. The results that we obtained from simulations are very encouraging. We plan to further validate the advantages of our approach through additional measurements.

## Acknowledgements

This work was supported in part by the National Science Foundation, the Office of Naval Research, by a research grant from the National Security Agency and a research donation from Intel Corp. Most development and testing were done on the computers of the National Center for Supercomputing Applications at the University of Illinois.

## References

[1] D.A. Padua, D.J. Kuck, and D.H. Lawrie, "High-speed Multiprocessors and Compilation Techniques", *IEEE Trans. on Computers*, C-29(9), 1980, pp. 763-776.
[2] D.A. Padua, and M.J. Wolfe, "Advanced Compiler Optimizations for Supercomputers", *Communications of the ACM*, 29(12), 1986, pp. 1184-1201.
[3] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Boston, 1988.
[4] B.R. Rau, and J.A. Fisher, "Instruction-level Parallel Processing: History, Overview and Perspectives", *Journal of Supercomputing*, 7(1), 1993, pp. 9-50.
[5] D.M. Lavery, *Modulo Scheduling for Control-Intensive General Purpose Programs*, PhD dissertation, University of Illinois at Urbana-Champaign, 1997.
[6] W.W. Hwu, S.A. Mahlke, W.Y. Chen, et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *Journal of Supercomputing*, 7, 1993, pp. 229-248.
[7] B.J. Smith, "A Pipelined, Shared Resource MIMD Computer", *Int. Conf. on Parallel Processing*, 1978, pp. 6-8.
[8] J.P. Laudon, *Architectural and Implementation Tradeoffs for Multiple-Context Processors*, PhD dissertation, Stanford University, 1994.
[9] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *Int. Symp. on Computer Architecture*, 1995, pp. 392-403.
[10] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors", *Int. Symp. on Comp. Arch..*, 1995, pp. 414-425.
[11] J. Tsai, and P. Yew, "The Superthreaded Archtecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation", *Conf. on Parallel Architectures and Compilation Techniques*, 1996, pp. 35-46.
[12] V.S. Krishnan, and J. Torrellas, "A Clustered Approach to Multithreaded Processors", *Int Parallel Processing Symp.*, 1998.
[13] P. Marcuello, A. Gonzalez, and J. Tubella, "Speculative Multithreaded Processors", *Int. Conf. on Supercomputing*, 1998, pp. 77-84.
[14] H. Akkary, and M.A. Driscoll, "A Dynamic Multithreading Processor", *Int. Symp. on Microarchitecture*, 1998, pp. 226-236.
[15] P.K. Dubey, K. O'Brien, K.M. O'Brien, and C. Barton, *Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading*, Res. Rep. RC 19928, IBM T. J. Watson Research Center, 1995.
[16] M. Prvulovic, M.J. Garzaran, L. Rauchwerger, and J. Torrellas, "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization", *Int. Symp. on Computer Architecture*, 2001.
[17] A. Zhai, C.B. Colohan, J.G. Steffan, and T.C. Mowry, "Compiler Optimization of Scalar Value Communication between Speculative Threads", *Int. Conf. on Arch. Support for Programming Languages and Operating Systems*, 2002.
[18] M.B. Girkar, *Functional Parallelism: Theoretical Foundations and Implementations*, PhD dissertation, University of Illinois at Urbana-Champaign, 1992.
[19] C.D. Polychronopoulos, *α-Coral: A New Multithreaded Processor Architecture, its Compiler Support, and Simulation of a multi-α-Coral Parallel System*, Project Proposal, CSRD, University of Illinois at Urbana-Champaign, 1997.
[20] G. Dimitriou, *Loop Scheduling for Multithreaded Processors*, PhD dissertation, University of Illinois at Urbana-Champaign, 2000.
[21] C. Moura, *SuperDLX – A Generic Superscalar Simulator*, ACAPS Tech. Memo 64, School of Computer Science, McGill University, 1993.