The KIT COSMOS Processor: A Preliminary Study on Transparent Software Prefetching via Dynamic Optimization

Kou Morita[1]

[1]Toshiba IT-Solutions Co., Japan

Toshinori Sato[2,3] Itsujiro Arita[2]

[2] Department of Artificial Itelligence

[3] Center for Microelectronic Systems

Kyushu Institute of Technology, Japan

Abstract

This paper proposes a method of cooperation between the dynamic optimization technique and simultaneous multi-threading (SMT) architecture. Recent studies on dynamic optimization reveal that it has a large potential for improving processor performance. This is because every program running on the processor can be optimized using profile information gathered on-the-fly, which is unavailable to compilers. On the other hand, the shipping of SMT processors is begining. They maintain several contexts simultaneously and improve the efficiency of their hardware resources. Thus, secondly threads exploit idle hardware resources which the primary thread cannot use. The dynamic optimization technique also benefits from SMT, since any overheads caused by the optimization are mitigated. This paper introduces a combination of dynamic optimization and SMT, and presents preliminary evaluation on transparent software prefetching, showing that processor performance is improved by up to 16.3%.

1 Introduction

In order to extract more instruction level parallelism (ILP), modern microprocessors rely on speculative execution or multi-path execution techniques. One of their drawbacks is that they execute useless instructions. Thus, they are justified only under the condition where a number of functional units are idle. Actually, these techniques have so far contributed to processor performance. However, deep speculation will fall in diminishing return as instruction issue width increases, since most of the instructions injected to the execution core might be worthless.

These considerations lead us to explore a new way to extract beneficial instructions. Idle functional units are efficiently utilized by multi-threading. An application and an optimizer are executed simultaneously on a multi-threaded processor. The optimizer improves performance of the application on-the-fly. We call this technique Condor [20]. Note that we will not claim that the total amount of ILP is increased by multi-threading. Instead, more ILP is extracted from the single application due to the dynamic optimization. This execution model is similar to simultaneous multi-threading (SMT) [24]. In addition, the shipping of SMT processors is begining [14]. Therefore, we investigate an architecture utilizing Condor on the top of the SMT processor, which we call the COSMOS processor.

The organization of this paper is as follows: Section 2 introduces the COSMOS processor. Sections 3 describes the dynamic optimization. Section 4 introduces the Condor architecture, and Section 5 presents primary evaluation results. Section 6 surveys related work. Finally, Section 7 concludes the paper.

2 COSMOS Processor

As explained above, the COSMOS processor is based on the SMT processor. It is important to consider the impact of hardware complexity on processor cycle time, since the SMT processor is basically a large-scale superscalar processor. In order to solve the problem, investigations have been widely performed in which the large processor is split into a number of smaller processing elements (PEs). Two candidate implementations are available for this purpose.

- On-chip multi-processor
- Clustered superscalar processor

The on-chip multi-processor consists of a number of independent processors, each of which is a PE. We are currently investigating the implementation of the COSMOS processor as a clustered superscalar processor. Figure 1 shows the block diagram of the COSMOS processor, which consists of one instruction supply mechanism and two instruction execution mechanisms (clusters). The number of the clusters is not

necessarily two. Interested readers can find more information in [22].

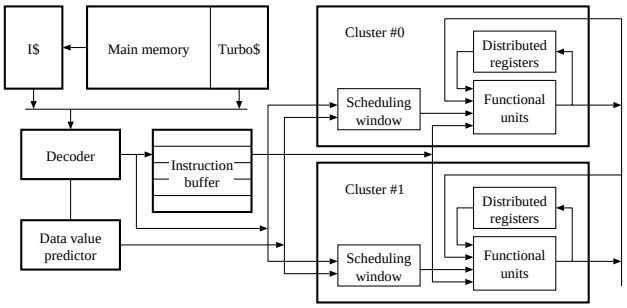

Figure 1: COSMOS processor

In this paper, we focus on Turbo cache, which is a key structure for Condor. Turbo cache is an on-chip SRAM and is mapped to a memory area which is not visible to the base architecture. It is used for the working area and for storing optimized binaries. Thus, the hardware-based Trace cache is replaced by the software-based Turbo cache. In addition, cache misses due to the interference between the optimized application and the optimizer threads are removed.

3 Dynamic Optimization

The dynamic optimization technique is different from the static optimization which compilers perform in that it optimizes a binary when its corresponding application program is executed [2]. Since the instruction set is maintained during the optimization, originally no special hardware is requested to support it. In addition, dynamic optimization does not translate input source code as in the manner of JIT compilers, and does not require any annotations in the input source code as in the manner of dynamic compilation systems. Dynamic optimization is only one step in the optimization process.

The dynamic optimization technique is effectively applicable to program sequences which compilers have difficulties in finding statically. For example, the dynamic technique makes it possible to optimize a binary including dynamically-linked libraries (DLLs) beyond procedure boundaries. Recent trends in advancing object oriented programs and in software distributions in the form of DLLs are favorable winds for dynamic optimization. Note that dynamic optimization will not replace static optimization, but there is a complementary relationship between them.

The dynamic optimization technique can utilize profile information gathered on-the-fly. Based on this information, frequently executed instruction traces are selected for optimization. It is expected that small portions of whole program dominate its execution time, and thus even optimizing only the small portions contributes to total performance. During the gathering of profiles, the program is interpreted by the dynamic optimizer. Note that the instruction set of interpretation is equivalent to the instruction set of the target machine. The original program is maintained unchangeably, and the processor transfers its control flow to a region in memory space where the optimized binary is stored.

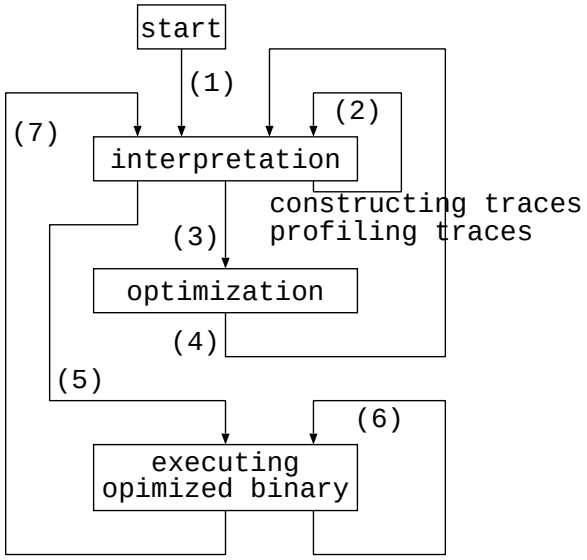

Figure 2: Dynamic optimization

Figure 2 explains dynamic optimization flow, which consists of seven steps.

1. The optimizer interprets the original binary.
2. It gathers profile information on-the-fly.
3. According to the profile information, a frequently executed trace is optimized. Meanwhile, execution of the original binary stops.
4. The optimizer resumes the execution after the optimization is completed.
5. When a branch to an optimized trace is detected, the processor transfers its control flow to the optimized trace.
6. The processor executes the optimized binary.
7. When a branch to an unoptimized portion is detected, the interpretation starts again.

4 Condor architecture

This section explains the Condor architecture, which is a software-controlled implementation of history-directed processor architecture [19]. Briefly, it combines the dynamic optimization technique and multi-threading, where an application program and an optimizer are executed simultaneously.

4.1 Overview of Condor

On Condor, instructions concerning the application and those concernning the optimizer are simultaneously issued to functional units. Functional units which are idle when executing the application are utilized for dynamically optimizing the application binary. It is found that desktop applications have a thread level parallelism (TLP) of at most 1.5 [12], and thus the SMT processor has enough contexts for Condor. Condor extracts more ILP from the optimized binary. When the ILP of the original binary is small, there are a number of idle functional units, which are used for the dynamic optimization. Otherwise, it is not necessary to optimize the binary.

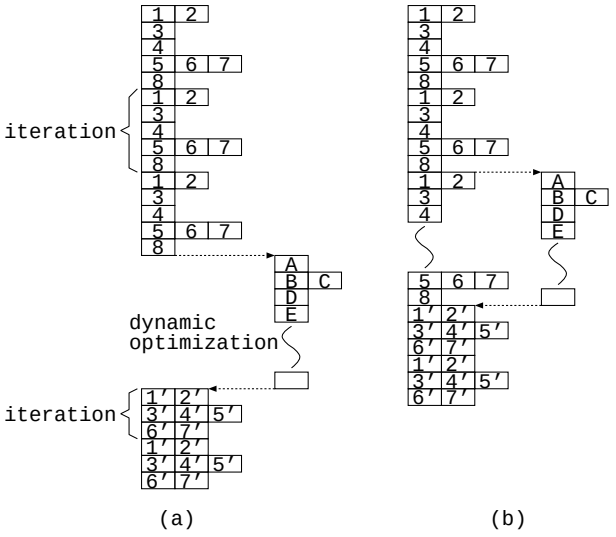

Figure 3: Condor dynamic optimization

Figure 3 shows the difference between the previously proposed dynamic optimization technique and Condor. In this case, a loop is the candidate for optimization. Figure 3(a) explains the previously proposed technique, and (b) describes Condor. The vertical axis denotes time. Each square corresponds to an instruction. The original application program consists of instructions `1,2,3...`, and the optimizing program (or Condor) consists of instructions `A,B,C....` After optimization, the original application becomes instructions `1',2',3'....`

In the case of the previously proposed dynamic optimizer denoted as (a), the condition initiating the optimization is that a loop frequency is beyond a threshold, for example 50 times. During optimization, the application stops executing. After the optimization is completed, the application again starts execution. Since the period for the optimization is an overhead for the application, it is important to decide which portions of the application should be optimized. This means that the threshold must be large.

In contrast, under the Condor architecture, the application continues to be executed during its optimization. Only idle functional units are used for the optimization. When the optimization is completed, the processor terminates the original binary and transfers its control flow to the optimized one. Hence, Condor is free from the overhead explained above. Since the overhead is negligible, it is possible to make the threshold for initiating the optimization relatively small. In addition, if it is found during optimization that the loop frequency is considerably small, the optimization can be discarded.

From the consideration above, it can be observed that the Condor architecture efficiently reduces the overheads for the dynamic optimization technique.

4.2 Profiling on Condor

As explained above, Condor utilizes profile information that has been dynamically gathered. Previously proposed dynamic optimizers have to obtain control of an application program when it starts. That is, executing the application is replaced by interpreting the application by the optimizers. One of the possible problems in this profiling is that interpreters are considerably slow.

In the Condor architecture, in contrast, profiling is performed by helper threads. When every instruction that is a candidate for profiling is encountered, it forks a helper thread. The helper thread is executed on the SMT and gathers profile information of the associated instruction. Its profile information is kept in the Turbo cache. The helper threads are inserted as some profiling tools, such as Shade [7] and ATOM [23], perform on binaries. In order not to diminish application thread performance, a resource allocation policy should monitor the resource utilization and determine if the helper threads could be triggered.

Dynamic profiling might be assisted by performance counters. Modern processors have the performance counters which collect information such as the number of instructions and cache misses. Horowitz et al. [13] propose a mechanism by which hardware is able to inform software of cache misses. Such a mech-

anism can reduce the overheads caused by profiling. It is also under consideration whether the performance counter will be utilized for profiling.

4.3 Optimization on Condor

When a Condor thread is triggered, it updates profile information and then checks the threshold for initiating the optimization. As explained above, it is possible to make the threshold relatively small since the overhead for the initiation is negligible.

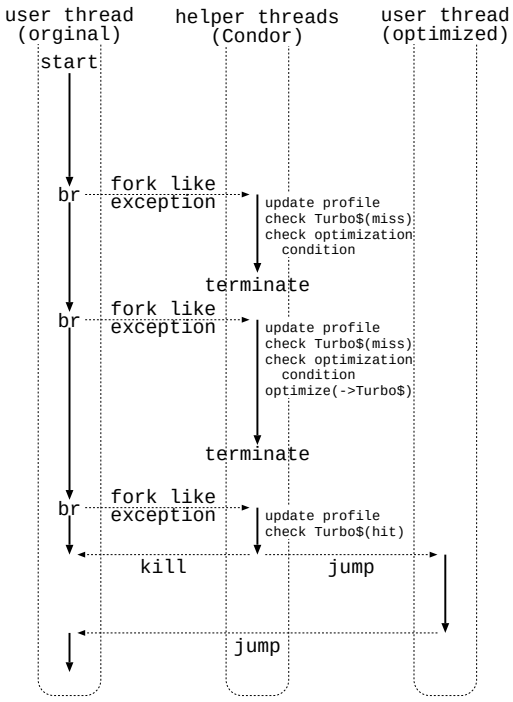

Figure 4: Execution flow of Condor

The Condor dynamic optimization is performed as described in Figure 4. It shows an example for profiling branch instructions. When every branch instruction is encountered, a helper thread is initiated in the manner of an exception handler [28] and an optimization process is triggered if its threshold condition is satisfied. The optimized binary is stored in the Turbo cache as explained above.

If the optimized binary is already in the Turbo cache when the helper thread is initiated, the control moves to the Turbo cache. In parallel, the execution of the original binary is squashed. Note that instructions following the branch instruction which initiates the helper thread are not retired until the helper thread is completed. When the control moves to any region which is not in the Turbo cache, the original binary restarts. The exchange of contexts between the original and optimized binaries can be easily handled by the synchronization mechanism implemented in the SMT processor [25].

4.4 Applications of Condor

This section explores application fields where Condor is effective. Condor is a general purpose instruction optimizer; hence, it can perform traditional optimization processes as modern compilers do. In addition, Condor can perform aggressive optimization based on profile information gathered on-the-fly as follows:

- Eliminating ineffectual branches [26, 27]: Some branch instructions are highly biased to one direction. Such branches can be replaced by unconditional jumps or can be removed completely. This optimization requires the insertion of some check instructions that verify the removal is correct. Because these check instructions can be moved from the original position and they can be condensed, basic block size is enlarged. In addition, it improves the utilization of branch prediction tables, increasing branch prediction accuracy. These contribute to the exploition of ILP.

- Transparent software prefetching: Based on profile information including cache miss histories, only frequently missed instructions can be selected for candidates of prefetching. In addition, prefetchings are initiated at the optimal point to hide miss penalties since Condor knows memory access latencies.

- Memory disambiguation: Serialization of a program caused by ambiguous memory dependences is one of the serious bottlenecks in modern superscalar processors. It is difficult for compilers to detect such dependences, especially pointer-intensive ones. Condor can easily detect them dynamically, and can eliminate unnecessary serializations of store and load instructions.

- Value prediction: Condor also contributes to data speculation based on value prediction. It is found that the efficiency of the value prediction on legacy binaries is quite smaller than that of the modern compiler optimization [21]. Thus, any cooperation between the value prediction and Condor is beneficial.

- Instruction packing: In SPECint95 benchmark, half of the instructions have operands of less than 16-bits [4]. Thus, in order to efficiently utilize

wide bitwidth functional units, the optimization of packing such instructions into a single SIMD-style instruction is useful. This is illustrated in the following example. There is an instruction in a loop, whose two operands are less than 16-bits almost all the time. If an ALU is 64-bits wide, four instructions can be packed into an instructions. In this case, Condor unrolls the loop four times, and then four identical instructions are replaced by a SIMD-style instruction. Since the bitwidth of operands is not determined until the instruction is encountered, compilers cannot do this optimization.

- Multi-path execution: Condor improves the efficiency of multi-path execution. Based on profile information, it can select branch instructions which cannot be predicted with high accuracy. These branches are converted dynamically to predicated instructions.

- Dynamic thread partitioning: Krishnan et al. have proposed software-based thread partitioning [16]. A single-threaded program is divided statically into multiple threads. Condor extends this scheme to perform thread partitioning dynamically with the help of the value prediction explained above.

Note that all the aggressive optimizations described above are not always applied to every application program. If the helper threads are generated in compile time, the compiler selects probably successful optimizations and provides them to the application binary.

It is true that compilers can utilize static profile information. However, dynamic profile information, which compilers cannot utilize, is more effective than static profile information due to the following reasons.

- Gathering profile information is tedious work for programmers. Thus, if it is automatically obtained when a program is in execution, programmers can use their time for other useful work.

- In the case of dynamic profiling, a training data set and an actual data set are equivalent. Therefore, significantly accurate profile information is obtained.

5 Preliminary Evaluation

We have already evaluated the potential of Condor on eliminating ineffectual branch instructions [26, 27]. In this paper, we will evaluate the potential on transparent software prefetching.

5.1 Methodology

We implemented a timing simulator using SimpleScalar/Alpha tool set (ver.3.0b) [5]. Our processor model is an out-of-order execution superscalar processor, and its configuration is summarized in Table 1.

Table 1: Processor configuration

Fetch Width	4 instructions
Branch Predictor	512-set 4-way set-associative BTB, 2048-entry bimodal, 8-entry RAS, 3-cycle miss penalty
Instruction Windows	16 entry instruction queue, 8 load/store queue
Issue Width	4 instructions
Functional Units	4 iALU's, 1 iMUL/DIV, 2 Ld/St's, 4 fALU's, 1 fMUL/DIV
FU Latency (total/issue)	iALU 1/1, iMUL 3/1, iDIV 20/19, Ld/St 2/1, fADD 2/1, fMUL 4/1, fDIV 12/12
Register Files	32 32-bit fixed point registers, 32 32-bit floating point registers
I-Cache	8K direct-mapped, 32B blocks, 6-cycle miss penalty
D-Cache	8K 4-way set-associative, 32B blocks, 2-port, 6-cycle miss penalty
L2 Cache	shared, 256K 4-way set-associative, 64B blocks, 48-cycle miss penalty

The SPEC2000 CINT benchmark suite is used for this study. Table 2 lists the benchmarks and the input sets. We use the object files provided by University of Michigan.

Table 2: Benchmark programs

program	input set
164.gzip	input.compressed 2
175.vpr	net.in arch.in
181.mcf	inp.in
186.crafty	crafty.in
255.vortex	lendian.raw
256.bzip2	input.random 2
300.twolf	test

First, we gather execution traces and determine the frequently missed load instructions. Then, we perform the optimization by hand using the profile information. We look for top 20 load instructions that frequently occur cache misses. This assumes

Table 3: %L1 data cache miss rates

Optimization	gzip	vpr	mcf	crafty	vortex	bzip2	twolf
Before	3.39	4.96	13.8	3.00	1.18	1.17	2.61
After	0.43	1.21	5.01	2.24	0.73	0.44	1.18

Table 4: Istructions per cycle

Optimization	gzip	vpr	mcf	crafty	vortex	bzip2	twolf
Before	1.81	1.35	1.07	1.30	1.56	2.09	1.35
After	1.92	1.57	1.21	1.32	1.57	2.21	1.36

that the hardware resources available to dynamic optimization are unlimited. Currently, we do not use any algorithms for prefetching. Instead, the simulator performs as if prefetch instructions are executed in the ideal position. Finally, we evaluate the optimization using the out-of-order execution 4-way superscalar processor.

5.2 Results

Table 3 summarizes L1 data cache miss rates before and after optimization, and Figure 5 shows the percent decrease in the miss rates. For five of seven programs, it is found that L1 data cache miss rates are decreased by over 50%. This is achieved by optimizing only 20 load instructions.

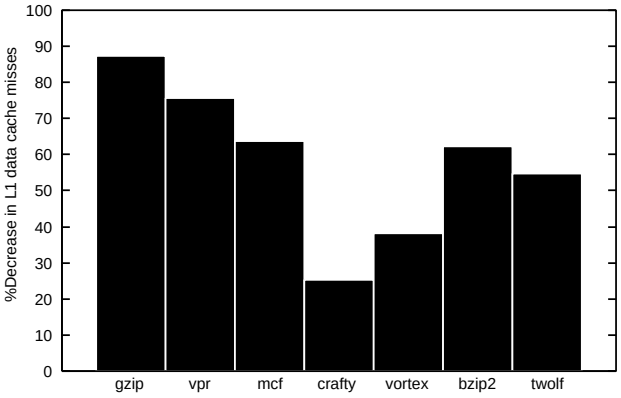

Figure 5: %Decrease in L1 data cache misses

Table 4 and Figure 6 show the contribution of the decrease in L1 data cache miss rates on processor performance. For `175.vpr` and `181.mcf`, it is found that ILP is improved by over 10%. It is interesting that the decrease in data cache misses does not always contribute to processor performance. For example, only 6.21% of ILP is increased in `164.gzip`, while approximately 90% of data cache miss is removed. For `186.crafty`, `255.vortex`, and `300.twolf`, the optimization on reducing data cache misses has little contribution on performance. ILP is improved by less than 2%.

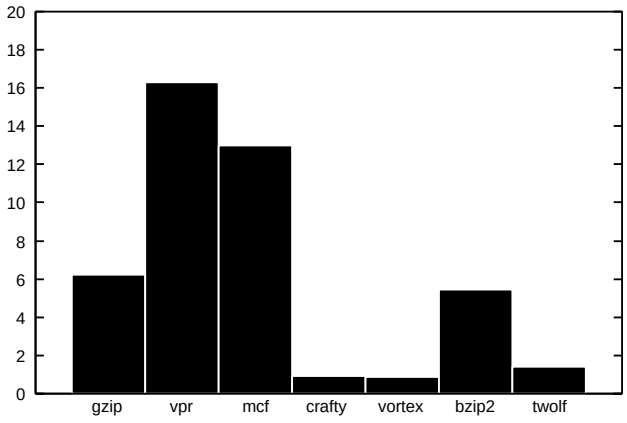

Figure 6: %Increase in ILP

From these results, we can confirm the potential of Condor on improving processor performace.

6 Related Work

Dynamo [2], DAISY [10, 11] and Code Morphing [15] are based on software technology. DAISY translates binary codes for an RISC machine into a VLIW machine. Similarly, Code Morphing processors translate binaries for the x86 instruction set into those for a VLIW instruction set. Condor does not change the instruction set during optimization. DAISY and Code Morphing gather profile information dynamically for translation [11,15]. Condor also utilizes dynamic profiling. One of the serious problems in DAISY is the

increased cache misses of the translated VLIW instructions [11]. On the other hand, Condor is free from the problem since it keeps instruction set during optimization.

Dynamo is a software optimizer and retains an instruction set during optimization. It also utilizes profile information gathered dynamically. Condor and Dynamo share these characteristics. However, Condor differs from Dynamo in regard to the point at which they initiate optimization. In the case of Dynamo, the application thread sleeps while it is optimized. On the other hand, Condor allows the application thread to continue during its optimization. The helper threads of Condor are executed simultaneously with the application thread. Due to this policy, it is important for Dynamo to reduce overheads caused by optimization. Thus, Dynamo is based on a heuristic in which an optimization is initiated after the corresponding trace appears more than 50 times. Condor is not constrained by the heuristic. It observes resource utilization and thus can decide efficiently when an optimization should be initiated. Lastly, Condor differs from these proposals in that it is supported by the SMT hardware, simplifying the management of each thread.

Merten et al. [18] propose the support of dynamic optimization using dedicated hardware which gathers profile information on-the-fly. This partly alleviates the drawbacks of the software-based dynamic optimization, but it does not rely on SMT hardware.

The Condor architecture proposed in this paper is strongly based on SMT architecture [24]. Originally, SMT was proposed to improve the throughput of processor resources by executing multiple applications in parallel. Thus, when only one application is executed on the SMT processor, it is difficult to exploit TLP. Following Tullsen et al., several extensions to SMT have been proposed.

Chappel et al. investigate a way of improving the performance of an application thread (a primary thread) by allowing subordinate microthreads to support the primary thread [6]. They call this mechanism Simultaneous Subordinate Microthreading (SSMT). For example, subordinate microthreads can assist the primary thread by prefetching data requested by the primary thread. Condor is different from the SSMT in the following ways. (1)SSMT requires a microcode RAM which holds the microthreads. On the other hand, Condor is placed on the main memory space. (2)SSMT does not optimize the application thread. In addition, it requires profile information in order to insert instructions which initiate the microthreads. Condor dynamically optimizes the application program using profile information gathered on-the-fly. Similar techniques in which the secondary threads assist the primary thread for the sake of data prefetching are studied in [1, 3, 8, 9, 17]. However, none of them performs dynamic optimization.

7 Summary

This paper has introduced Condor architecture which can extract more ILP by increasing beneficial instructions. The key to this architecture lies in the cooperation between hardware and software. An application thread and helper threads which optimize the application dynamically are executed simultaneously on an SMT processor. Idle functional units are efficiently utilized and thus overheads caused by the optimization can be eliminated. These characteristics alleviates the drawbacks of the previously proposed dynamic optimization techniques. Our preliminary evaluation on transparent software prefetching shows that processor performance is improved by up to 16.3%.

Acknowledgments

This work is supported in part by Grant-in-Aid for Scientific Research (No.13558030) from the Japan Society for the Promotion of Science.

References

[1] M.Annavaram, J.M.Patel, and E.S.Davidson, "Data prefetching by dependence graph precomputation," 28th Int. Symp. on Computer Architecture, 2001.

[2] V.Bala, E.Duesterwald, and S.Banerjia, "Transparent dynamic optimization," Technical Report HPL-1999-77, HP Laboratory, 1999.

[3] R.Balasubramonian,S.Dwarkadas, D.H.Albonesi, "Dynamically allocating processor resources between nearby and distant ILP," 28th Int. Symp. on Computer Architecture, 2001.

[4] D.Brooks, M.Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," 5th Int. Symp. on High-Performance Computer Architecture, 1999.

[5] D.Burger and T.M.Austin: "The SimpleScalar tool set, version 2.0," ACM SIGARCH Computer Architecture News, vol.25, no.3, 1997.

[6] R.S. Chappel, J. Stark, S.P. Kim, S.K. Reinhardt, and Y.N. Patt, "Simultaneous subordinate microthreading (SSMT)," 26th Int. Symp. on Computer Architecture, 1999.

[7] R.F.Cmelik and D.Kepple, "Shade: a fast instruction-set simulator for execution profiling," Technical Report UWCSE93-06-06, Dept. of Computer Science and Engineering, Univ. of Wasington, 1993.

[8] J.D.Collins, H.Wang, D.M.Tullsen, C.Hughes, Y-F.Lee, D.Lavery, and J.P.Shen, "Speculative precomputation: large-range prefetching of delinquent loads," 28th Int. Symp. on Computer Architecture, 2001.

[9] M.Dubois and Y.H.Song, "Assisted execution," Technical Report CENG-98-25, University of Southern California, 1998.

[10] K.Ebcioglu and E.Altman, "DAISY: dynamic compilation for 100% architecture compatibility," 24th Int. Symp. on Computer Architecture, 1997.

[11] K. Ebcioglu, E.R. Altman, S. Sathaye, and M. Gschwind, "Execution-based scheduling for VLIW architectures," 5th Int. Euro-Par Conf., 1999.

[12] K.Flautner, R.Uhlig, S.Reinhardt, and T.Mudge, "Thread-level parallelism and interactive performance of desktop applications," 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, 2000.

[13] M.Horowitz, M.Martonosi, T.C.Mowry, and M.D.Smith, "Informing memory operations: memory performance feedback mechanisms and their applications," ACM Transactions on Computer Systems, vol.16, no.2, 1998.

[14] Intel Corporation, "Introduction to Hyper-Threading technology", White paper, 2001.

[15] A.Klaiber, "The technology behind Crusoe processors," White Paper, Transmeta Corp., 2000.

[16] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," IEEE Transactions on Computers, vol.48, no.9, 1999.

[17] C-K.Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," 28th Int. Symp. on Computer Architecture, 2001.

[18] M.C. Merten, A.R. Trick, C.N. George, J.C. Gyllenhaal, and W-m.W. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," 26th Int. Symp. on Computer Architecture, 1999.

[19] T.Sato, "History directed processor architecture," Doctoral thesis, Kyoto University, 1998.

[20] T.Sato and I.Arita, "The KIT COSMOS processor: introducing CONDOR," Int. Conf. on Parallel and Distributed Processing Techniques and Applications, 2000.

[21] T.Sato, A.Hamano, K.Sugitani, and I.Arita, "Influence of compiler optimizations on value prediction," 9th Int. Conf. on High Performance Computing and Networking Europe, 2001.

[22] T.Sato, T.Yamamoto, and I.Arita, "The KIT COSMOS processor: a low-complexity superscalar processor," Int. Journal of Computer & Information Science, vol.2, no.4, 2001.

[23] A.Srivastava and A.Eustace, "ATOM: a system for building customized program analysis tools", Conference on Programming Language Design and Implementation, 1994.

[24] D.M.Tullsen, S.J.Eggers, and H.M.Levy, "Simultaneous multithreading: maximizing on-chip parallelism," 22nd Int. Symp. on Computer Architecture, 1995.

[25] D.M.Tullsen, J.L.Lo, S.J.Eggers, and H.M.Levy, "Supporting fine-grained synchronization on a simultaneous multithreading processor," 5th Int. Symp. on High-Performance Computer Architecture, 1999.

[26] T.Yamamoto, T.Sato, and I.Arita, "The KIT COSMOS processor: eliminating ineffectual branch instructions via concurrent dynamic optimization," COOL Chips IV, 2001.

[27] T.Yamamoto, K.Morita, T.Sato, and I.Arita, "The KIT COSMOS processor: an application of multi-threading for dynamic optimization," Int. Conf. on Parallel and Distributed Processing Techniques and Applications, 2002.

[28] C.B.Zilles, J.S.Emer, and G.S.Sohi, "The use of multithreading for exception handling," 32nd Int. Symp. on Microarchitecture, 1999.