Using Pre-Execution and Helper Threads for Speeding Up Data Intensive Applications

Ákos Dudás and Sándor Juhász

***Abstract*—Pre-execution is a new technique used in conjunction with simultaneous multithreading or multi-core CPUs to reduce memory latency. Executing a slice of a program in a software or hardware thread ahead of the normal execution resolves memory addresses and prefetches data into the caches. By doing so the latency of memory reads is reduced in the main thread. Data intensive applications can benefit from pre-execution even if thread level parallelism is not available because of shared (software) resources. Despite the simplicity of the idea several factors have to be considered when putting this principle into practice. This paper applies and customizes pre-execution to a particular problem of hash table based data transformation and through this example provides a parameterized software pre-execution algorithm which can be applied to arbitrary programs and executed on everyday hardware. The most important factor to be considered is to continuously keep an optimal temporal distance between the pre-worker and the main thread, which should be implemented with introducing minimal amount of control and communication dependencies between them. This paper presents a mechanism for attaining this goal.**

***Index Terms*—data prefetch, pre-execution, performance, multi-threading**

I. Introduction

DATA intensive applications have long been suffering from the relatively slow speed of main memories compared to the performance of current CPUs. Over the years there has been many techniques integrated into CPUs that directly or indirectly try to address this problem. These techniques include cache memories integrated into the CPU and onto the same die; speculative execution and branch prediction; and hardware data prefetch methods. They are all available in current CPUs and try to mask the effect of memory latency without any additional effort required on behalf of the programmer.

With the appearance of simultaneous multithreading and multi-core CPUs there has been a vast number of software and hardware techniques proposed in the literature; all built around the same idea called *pre-execution* [1]–[8] (terminology also includes assisted execution, prefetching helper threads, microthreading, speculative precomputation). While the proposed techniques differ in some aspects they are all based on the same idea: *hiding* memory latency by prefetching the data with the help of threading. The helper thread, responsible for the prefetching, must "anticipate" what memory address will be referenced in the future by the main thread. This helper thread could be for example a microthread automatically started by hardware [1]; could be a software thread extract of the original program started automatically or manually [3], or speculative extract (slice) of the original algorithm [9], [10].

Manuscript received February 15, 2011; revised March 21, 2011.

The authors are with the Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Hungary. e-mail: {akos.dudas,juhasz.sandor}@aut.bme.hu

Having a simultaneous multithreading (SMT) or a true multi-core CPU at our disposal we can execute multiple threads on a single CPU without (significant) performance penalty. Although in both types of CPUs the (logical - in case of SMT) cores compete for some of the system resources (including the main memory and the last level of shared cache, and even execution units in SMT), the level of instruction level parallelism increases: while one of the executing threads stalls on a memory read operation the other thread can still execute, making better use of the CPU and reducing the overall execution time.

In this paper we focus on data intensive applications. The main characteristics of such problems is that they access and process data residing on disc or memory, and the performance bottleneck is the access time to this data. Data intensive applications are quite frequent. A simple web search involves gigabytes if not terrabytes of data [11]; other related ares include medical imaging, astronomical data analysis, weather simulation, or analyzing data collected during physical experiments [12].

Consider the following data intensive task. A set of items (billions of records) are pushed through a pipeline of transformation steps, one of which is encoding with the use of a dictionary. This problem (data transformation) is clearly data intensive: the transformation is a series of simple computations and a lookup/insert in the dictionary. The dictionary is not static (it is expanded with a new record when a previously unknown item emerges) the accesses to the dictionary must be synchronized which makes parallelization harder and causes overhead. Instead of true parallelization we use pre-execution to increase its performance.

In more general terms we focus our attention to data intensive applications bounded by one constraint: there is a central resource which makes it unfeasible to efficiently parallelize the algorithm. The main contribution of this paper is providing a general template for software pre-execution. The template we propose is a simple yet powerful algorithm with two tunable parameters having a performance gain of 5-15% even on a standard desktop computer. Through manual optimization steps we aim at reaching the full potential of pre-execution.

The rest of this paper is organized as follows. Section II gives a short survey of related literature. Section III explains how pre-execution techniques can be used to fight cache misses, a central issue in data intensive applications; Section IV details the idea and implementation considerations of pre-execution. The empirical evaluation of these techniques is given in Section V. We conclude in Section VI with a brief discussion about the future of this technique.

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

II. Related literature

Luk in [4] argues that irregular memory access patterns (responsible for increased memory latencies and inefficient hardware prefetch) cannot be predicted only through executing the code. This is the basic idea of pre-execution.

The works under the umbrella of pre-execution (assisted execution, subordinate microthreading, etc.) can be categorized as hardware and software solutions. Hardware solutions require the modification of the CPU or the entire computer architecture while software solutions can be used on any currently available CPU.

Hardware solutions [1], [2], [4], [6], [8], [13], [14] require minor or major modifications to current CPU architectures; usually along with compiler support. Some of these ideas, such as the one discussed in [13] propose compiler support to automatically find the code segments which can be optimized and sped up by a second thread dedicated to speculative prefetching.

Zilles et al. in [8] also proposed speculative execution. Speculation in this context means that the original algorithm is modified (manually or automatically) and certain instructions are removed. The goal is to speed up the execution at the cost of potentially arriving at a wrong result; which is discovered no later then the main execution thread completing the calculations. This idea is the data flow analogue of branch prediction used in control flow.

Chapell et al. in [1] presented another hardware-based solution. They proposed microthreads implemented in the CPU. These microthreads (series of micro-operations) are responsible for, when triggered, prefetching the data. Dubois in [2] proposes a similar solution called assisted execution where nanothreads help the main thread. This proposition also requires hardware support for the nanothreads to be able to use the main threads registers and memory.

The advantage of software approaches [3], [5], [7], [9] compared the hardware based solutions discussed above is that they can be evaluated without simulation. They can be implemented right away and need no costly manufacturing or modification of existing hardware. Instead they are ready to be prototyped and executed on any current CPU and the performance gain can be measured in terms of reduced execution time or reduced cache miss count for example.

The most promising work in this area is by Kim et al. [3]. Not only did they use compiler support for automatic helper thread creation but also used CPU performance counters to find the hotspots of the algorithm worth exploring and working with. Their work is quite comprehensive but it can be argued that manual fine-tuning can achieve even better wall-clock speedups. Similar work has been published by Ro and Gaudiot [15] (they also require hardware support for triggering the p-thread - pre-execution thread) and by Yonghong et al. [16].

The work of Malhotra and Christos [5] extended the C++ STL library with pre-execution capabilities demonstrating that even general-use software can benefit from pre-execution.

For a survey and a taxonomy of data prefetch mechanisms (including pre-execution techniques) see [17].

Inspired by the works presented in this section our goal is to taking software pre-execution to the next level and providing a template which can be applied to any data intensive application. Our purpose is pushing the boundaries of pre-execution through manual optimization.

III. Cache misses

In modern data intensive applications memory latency is a real concern [18]–[21]. Cache memories are integrated into the CPU for this reason; they provide fast access to data accessed in the past. It is also argued by Abraham et al. in [22] that a small fraction of instructions are responsible for most of the cache misses. The aim of pre-execution is precisely finding these instructions and triggering an early prefect of the memory addresses they require. Once they are prefetched into the cache the next time the data is requested it will be served by the cache saving the CPU from severe performance penalty.

The memory addresses cannot be efficiently predicted in complex algorithms [4]. When iterating through an array for example it is easy to predict the next address and software and hardware prefetch mechanisms make due. The goal of pre-execution on the other hand is to handle arbitrary memory access patterns, such as finding items in a hash table based dictionary in our case.

Pre-execution dispatches two copies of the same algorithm: a "main" version and the "pre-execution" version which is a slight modification of the algorithm (it commits no changes into the main memory). Any data one thread loads into the cache is accessible to the other thread (supposing they share the same L2 or L3 cache). When the pre-execution thread works ahead of the main thread (see Figure 1) it paves the way in front of the main thread. Of course this effect has a limited scope; if the pre-execution thread prefetches data too early, the data might be replaced in the cache by the time the main thread arrives at to point to be able to use them.

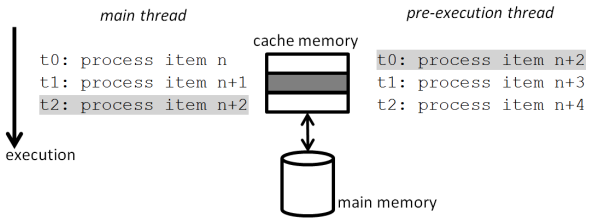

Fig. 1. The main thread and the pre-execution thread working side-by-side.

IV. Software pre-execution

In this paper we use a strictly software approach with a perfect predictor: the algorithm itself. The justification for the software approach is the relative short time it takes to prototype the solution. Every high-level programming language has ways of dynamically creating threads and managing communication and synchronization between them. Hence it is easy to develop and test.

A. The general idea (N)

A common approach to implementing of pre-execution (e.g. in [5], [7]) is using trigger points in the algorithm when an early prefetch can be issued (see Figure 2). This request is dispatched to a new thread ([8]) or is queued and carried out by a dedicated worker thread [7].

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

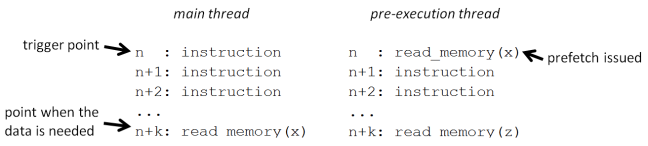

Fig. 2. Trigger point of dispatching a prefetch request.

In this paper we propose a different approach for the following reasons. The trigger point has to be specified manually, which is not easy; and if it is placed not early enough the prefetch happens too late. Also in certain cases it is hard to issue an early prefetch, say 100-200 instructions before the data is needed. The interval between the point where the memory address is calculated and the point of the actual memory read is usually a few of instructions only.

We resolve this situation by using a worker thread which proceeds (mostly) independently from the main thread and executes exactly the same instructions. This pre-execution thread calculates the memory addresses early and issues a load instruction. As the two threads operate the same way there is no need for communication between them to find out what to load. The pre-execution thread is a perfect predictor this way. We just have to make sure that the pre-execution thread is ahead of the main thread at all times (will be discussed below). The biggest advantage of this approach that it is applicable to *any* algorithm.

The methodology we used is as follows. We remind the reader that this is only the first, naïve implementation. Throughout this section we only provide a starting point to prepare the discussion of more advanced versions.

1) Given a single-treaded algorithm we strip it down to the most critical parts that actually performs the operations on the data. This slice [8] of the program is what we will work on.[1]
2) This slice (algorithm excerpt) is examined and in case there is any modification it commits to memory these operations are (manually) removed from the pre-execution thread thus only one of the threads is allowed to make permanent changes.
3) The algorithm is amended with shared variables that register the index of the item being processed. This information is used for synchronization between the threads.
4) Finally using the operating systems threading capabilities the processing is started in two parallel threads; one is the main thread (allowed to commit results) and the other one is the pre-execution thread which cannot perform write operations to the shared memory. (This thread is allowed to have private variables that can be modified locally but are invisible for the main thread.)

Both threads work on the same data set from the very first item to the last one. They both process every single item. As the threads are not guaranteed to run with the same pace some level of control is necessary. By introducing index counters (i.e. which item are the threads currently processing) we can keep track of progress of the threads and intervene if required.

If the main thread is far behind the data the pre-execution thread loads into the cache could get overwritten by other data before it is referenced by the main thread; on the other hand if the thread are to close, the pre-execution thread has little chance of making the prefetch early enough and becomes a waste of system resources making no contribution to the execution time.

In order to keep the temporal distance optimal, we look at the index counters in both threads. If the main thread seems to be too far behind the pre-execution thread is paused for a short time. Pause means yielding the CPU before the end of the current time slot and implicitly asking the scheduler to run the main thread instead. On the other hand, if the main thread sees, that the other thread is not far ahead enough, it will renounce of its time slice for the benefit of the pre-execution thread.

A note on implementing the index counters. The threads share the index counter variables (accessible in both threads). When this variable is written the CPU must invalidate any copy of the same variable cached by other cores of the CPU. If these variables share the same cache line every write operation to this variable will invalidate the entire cache line. This is called cache-trashing. To overcome this potential issue each counter is stored on 64 bytes, equal to the size of the cache line, but only the first four bytes are used for storage. This makes sure no two counters share the same cache line. Since writing an integer is atomic on x86 (and compatible) architecture no other synchronization is necessary other then declaring the variable volatile. (We understand and accept the fact that this does not guarantee proper consistency but neither do we require it. A “not to old” value of this index counter is good enough as long as the overhead of complete thread safe access locks are eliminated.)

B. Jumping in the data set (J)

Instead of suspending the main thread it makes more sense always to let it run without any interruption and control only the pre-execution thread. Controlling the pre-execution thread is usually done by triggering it at specific points points in the algorithm or when specific hardware events occur. We instead propose that the pre-execution thread is run continuously as before, but moved ahead of the main thread by a specified distance. Exploiting the possibility of “jumping” back and forth in the data set (i.e. being able to access and arbitrary item in the data set) the pre-execution thread can be instructed to move ahead in the dataset when it falls behind (its index counter is compared to the main threads index counter), or go back when it goes too far ahead.

Effectively this means that the pre-execution thread may skip certain items and process other ones more than once. Neither is a problem in terms of performance. It goes without saying that skipping items has no effect on performance. If an item is processed more than once the CPU is kept busy but not at the cost of wasting memory resources. Memory loads will either be served by the cache (as they have been preloaded during the last pass) or by the main memory if the data has been evacuated from the cache in the meantime. In the latter case the load will not go to waste as the main thread will still need the data later.

[1]It is very common in data-processing algorithms that the dataset is iterated through once and the items are evaluated one-by-one. In such constellation it is very easy to pinpoint the part of the algorithm we can accelerate: it is the loop which traverses the data set [3].

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

The worker thread is never allowed to get behind the main thread. The amount of items to jump ahead of the main thread is a configurable parameter of the algorithm. Skipping far ahead could cause items to be long evicted from the cache when the main thread accesses it; not keeping enough distance could allow the main thread to take over. Finding an optimal value is important. Using a simple model for cache behavior an upper bound for this parameter can be calculated by dividing the cache size by the size of the data items prefetched.

C. Workahead-set (W,Wb)

The next - and last - enhancement to our algorithm is the use of workahead-sets. Here we make use of the fact that there is no need for intervention in the pre-execution thread after every processed item. We can let the process go unattended for a while and check it only, if it finished processing e.g. a thousand items. Infrequent synchronization makes better utilization of the CPU and reduces the cost of control.

Of course the set size (i.e. how long we let the pre-execution thread run without supervision) must be determined and chosen wisely.

This concept (W) is called workahead-sets in [7]. It is suggested in that work that traversing the workahead-set backwards (Wb) may resolve issues when both threads work with the same item.

D. Summary

Table I summarizes the features and the characteristics of the algorithms presented above.

TABLE I
SUMMARY OF THE FEATURES OF THE PRE-EXECUTION ALGORITHMS. THE NAME OF EACH VERSION IS DERIVED FROM THE FIRST LETTER OF THE FEATURE WHICH DESCRIBES IT MOST; E.G. W MEANS WORKAHEAD-SET.

Alg.	Waits	Workahead-set	Forward
N	yes	no	yes
J	no	no	yes
W	no	yes	yes
Wb	no	yes	no

V. EVALUATION

In this section we present the evaluation of the algorithms discussed so far. As explained in Section I a data intensive application, which is the case study for this paper, is data transformation by the use of a central dictionary. The dictionary is implemented as a hash table. To show the effectiveness of the presented technique three different hash table implementations were tested. The hash tables are Google's sparsehash and a two custom optimized implementations (an open- and a bucket hash table).

The processing algorithms are implemented in C++ with careful optimization and compiled for 32 bit with full optimization configuration with Microsoft Visual Studio 2008. The test were executed on an Intel Core i7-920 CPU (4 physical cores and HyperThreading) @ 2.66 GHz with 6 GB RAM and Windows 7 64 bit.

In every test scenario the same amount of data was processed and every test case was repeated 5 times and the results were averaged.

A. HyperThreading and multi-core CPU

Intel's HyperThreading technology enables a single physical core to appear as two logical cores and allows the operating system to schedule two threads on the single core. Some components of the core are duplicated but the execution units are not. Such SMT CPU is not the equivalent of a dual-core CPU.

However this SMT solution may also has benefits. In the Core i7 CPU the cores have dedicated L2 caches (512kB each) and a shared L3 cache (8MB). In a single-core with HyperThreading setup the logical CPUs share the L2 cache and the data in it. In a dual-core no HyperThreading setup the cores share only the L3 cache and have distinct L2 caches. At the same time our implementation of pre-execution is not speculative, that is, the pre-execution thread executes almost the same code as the main thread. This means it uses the CPU quite heavily, which may be a problem with SMT. The cost-benefit is not clear in this case.

Both setups were used in our evaluation: single-core with HyperThreading and dual-core without HyperThreading. (On a technical note the CPU allows the user to enable or disable HyperThreading and to disable cores in the CPU.)

B. Pre-execution vs. single threaded approach

The first batch of tests will determine if pre-execution as a whole makes sense. The four different pre-execution implementations (discussed in Section IV) are compared to a baseline single threaded algorithm.

Figure 3 shows the measured execution times of the algorithms when a single core was enabled in the CPU with HyperThreading turned on. As expected the *N* version (one with wait instructions) performs quite badly. It might be surprising at first but it is necessary to understand that the expected (and promised) performance gain does not come at zero cost. A naïve pre-execution implementation makes things worse. Removing the wait instructions from the implementations and using the jumpahead parameter instead (J) resolves this problem. Adding the workahead-sets to the picture (W, Wb) and we got a modest 2 to 13 percent gain in performance; except for the third hash table google sparsehash.

Figure 4 displays the results of the same test repeated on the same CPU but now with two physical cores enabled and HyperThreading turned off. The picture has changed dramatically. The J, W and Wb pre-execution algorithms are superior by a wide margin in each test case.

The difference once again is SMT or true multi-core CPU. There cannot be any doubt that pre-execution is a valid and promising technique. From the results above it seems like that SMT is good for pre-execution but true multi-core CPUs are even better. Since such CPUs dominate the market nowadays this is not a real concern.

It must also be noted that the advantage of SMT over multi-core CPUs, discussed in Section V-A, hold true, however in the Intel Core i7 CPU the L2 cache shared by the logical cores of a single core is only 512kB. The L3 cache shared by all cores on the other hand is 8MB. Hence the small fraction of data which is preloaded into the L2 cache when executed on an SMT processor does not outweigh the performance loss of the two threads sharing CPU resources.

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

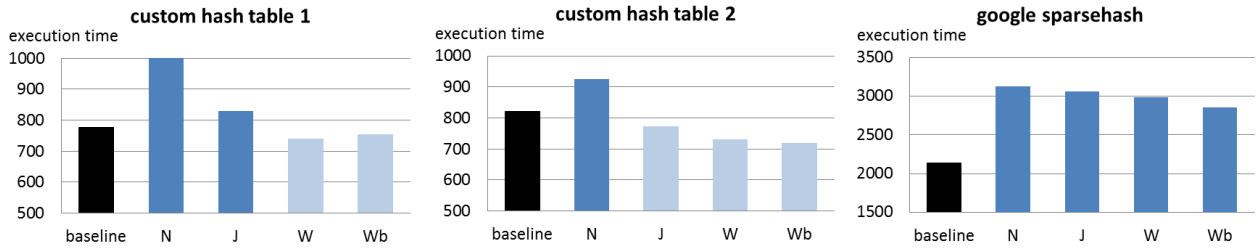

Fig. 3. All pre-execution versions with all three hash tables executed on a single core with HyperThreading enabled. The first bar in each figure is the baseline single threaded application. The lighter colored bars indicate shorter execution time than the baseline.

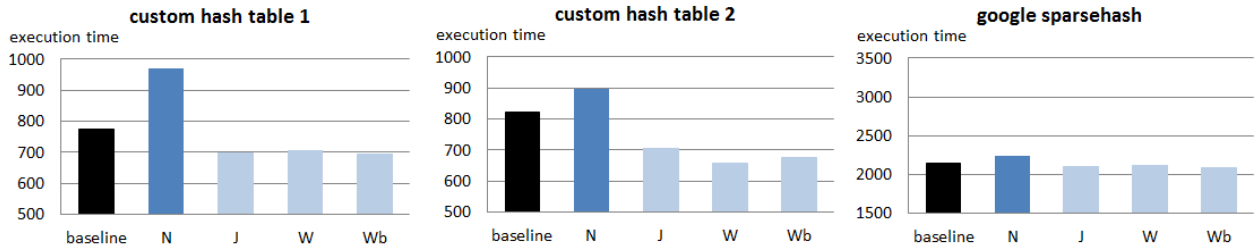

Fig. 4. All pre-execution versions with all three hash tables executed on two cores with HyperThreading disabled. The first bar in each figure is the baseline single threaded application. The lighter colored bars indicate shorter execution time than the baseline.

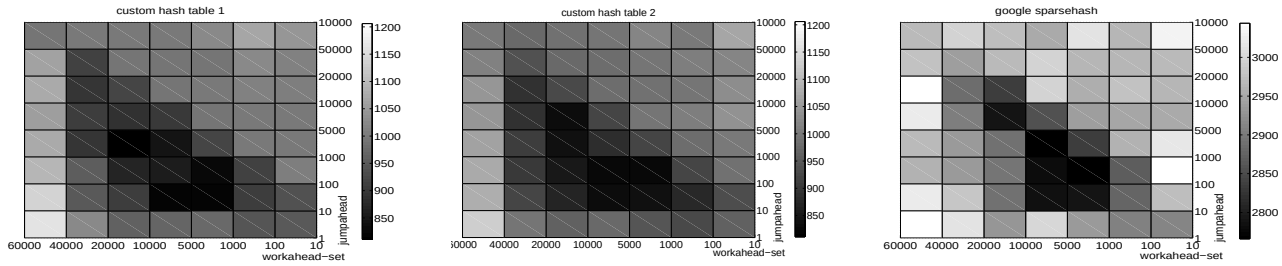

Fig. 5. The execution time as a function of the workahed-set size (horizontal axis) and jumpahead distance (vertical axis). The darker the shade is the shorter the measured execution time is.

C. Jumpahead and workahead-set size

The last question which needs to be answered is now to choose the workahead-set size (Section IV-C) and the jumpahead parameter (Section IV-B). Figure 5 plots the execution time as a function of both parameters. The vertical axis is the jumpahead parameter and the horizontal is the workahead-set size. The darker the shade is the shorter the execution time is.

There is a clear pattern discoverable in all three figures. But first the parameter ranges have to be explained. The jumpahead distance can obviously start from 0 which would make no sense (it would mean consider the item the main thread is currently working on); the upper limit is how many items fit into the cache before starting to overwrite them. In case of 8MB L3 cache and prefetched data size of 32 bytes it is 256.000 items. We chose 100.000 as the maximum for this parameter.

The workahead-set size should be no less than 1. There is no theoretical upper limit; it is a cost-benefit decision. More synchronization (lower value) is overhead while allows more control.

From Figure 5 we can see that there seems to be a connection between the two parameters. The lower the workahead-set size it the smaller jumpahead distance we should choose. The explanation is that more frequent synchronization allows fine-grained control. If we choose infrequent control a larger safety zone (bigger temporal distance) should be used between the main thread and the pre-execution thread.

VI. Conclusion and Discussion

In this paper we presented a technique called pre-execution which promises hiding memory latency by issuing early prefetch requests of memory addresses. When such algorithm is executed on multi-core or SMT CPUs performance gain can be obtained.

Through the case study of data transformation it has been presented that data intensive applications can benefit from applying this technique. The pure software solution we propose uses a dedicated thread which is executed parallel to the main thread; and they both execute almost the same code. This method is easy to implement as it requires minimal modifications to the code base, but have the advantage of general applicability.

We demonstrated that simple implementations of this idea may result in performance loss rather than gain which is not to hold against the technique but rather the implementation. Frequent synchronization should be avoided and the threads, though should be supervised, must not be suspended.

It was shown that using workahed-sets and a jumpahead distance the pre-execution thread can be kept ahead of the

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

main thread. In the future we are planning implementing an automated algorithm which configures these parameters in runtime by monitoring the threads.

ACKNOWLEDGMENT

This work is connected to the scientific program of the "Development of quality-oriented and cooperative R+D+I strategy and functional model at BUTE" project. This project is supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

REFERENCES

[1] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (SSMT)," *International Symposium on Computer Architecture*, vol. 27, no. 2, p. 186, 1999.

[2] M. Dubois, "Fighting the memory wall with assisted execution," in *Proceedings of the 1st Conference on Computing Frontiers*, Ischia, Italy, 2004, pp. 168–180.

[3] D. Kim, S. S.-w. Liao, P. H. Wang, J. D. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, "Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2004, pp. 27–38.

[4] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 2, pp. 40–51, Jun. 2001.

[5] V. Malhotra and C. Kozyrakis, "Library-based Prefetching for Pointer-intensive Applications," 2006.

[6] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," *ACM SIGPLAN Notices*, vol. 33, no. 11, p. 115/126, 1998.

[7] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah, "Improving database performance on simultaneous multithreading processors," in *Proceedings of the 31st International Conference on Very Large Data Bases*, Trondheim, Norway, 2005, pp. 49–60.

[8] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 2, pp. 2–13, 2001.

[9] M. Cintra and D. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *PPoPP03 Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2003, pp. 13–24.

[10] C. B. Zilles and G. S. Sohi, "Understanding the backward slices of performance degrading instructions," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 172–181, 2000.

[11] R. E. Bryant, "Data-intensive supercomputing: The case for disc," 2007.

[12] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger, "Data management in an international data grid project," in *Grid Computing GRID 2000*, ser. Lecture Notes in Computer Science, R. Buyya and M. Baker, Eds. Springer Berlin / Heidelberg, 2000, vol. 1971, pp. 333–361.

[13] J. G. Steffan and T. C. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pp. 2–13.

[14] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *roceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, Dallas, Texas, United States, 1998, pp. 59–68.

[15] W. W. Ro and J.-L. Gaudiot, "Spear: a hybrid model for speculative pre-execution," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE, 2004, pp. 75–84.

[16] Y. Song, S. Kalogeropulos, and P. Tirumalai, "Design and implementation of a compiler framework for helper threading on multi-core processors," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE Computer Society Washington, DC, US, Sep. 2005, pp. 99–109.

[17] S. Byna, Y. Chen, and X.-H. Sun, "A Taxonomy of Data Prefetching Mechanisms," in *Proceedings of the 2008 International Symposium on Parallel Architectures, Algorithms, and Networks (i-span 2008)*. Sydney: IEEE, May 2008, pp. 19–24.

[18] A. Glew, "MLP yes! ILP no!" in *Wild and Crazy Ideas Session, 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[19] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=216588

[20] J.-A. M. Anderson, J. Dean, J. E. Hicks, C. A. Waldspurger, and W. E. Weihl, "Method for inserting memory prefetch operations based on measured latencies in a program optimizer," 1997.

[21] T. Karkhanis and J. E. Smith, "A Day in the Life of a Data Cache Miss," in *Workshop on Memory Performance Issues*. In Workshop on Memory Performance Issues, 2002.

[22] S. Abraham, R. Sugumar, D. Windheiser, B. Rau, and R. Gupta, "Predictability of load/store instruction latencies," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*. Austin, TX , USA: IEEE Comput. Soc. Press, 1993, pp. 139–152.

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)