CacheFlow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading

Costas Kyriacou[1], Paraskevas Evripidou[2], and Pedro Trancoso[2]

[1] Computer Engineering Depart., Frederick Institute of Technology, Cyprus
[2] Department of Computer Science, University of Cyprus, Cyprus
{cskyriac,skevos,pedro}@cs.ucy.ac.cy

Abstract. With Data Driven Multithreading a thread is scheduled for execution only if all of its inputs have been produced and placed in the processor's local memory. Scheduling based on data availability may be used to exploit short-term optimal cache management policies. Such policies include firing a thread for execution only if its code and data are already placed in the cache. Furthermore, blocks associated to threads scheduled for execution in the near future, are not replaced until the thread starts its execution. We call this short-term optimal cache management policy the CacheFlow policy.
Simulation results, on a 32-node system with CacheFlow, for eight scientific applications, have shown a significant reduction in the cache miss ratio. This results in an average speedup improvement of 18% when the basic prefetch CacheFlow policy is used, compared to the baseline data driven multithreading policy. This paper also presents two techniques to further improve the performance of CacheFlow: conflict avoidance and thread reordering. The results have shown an average speedup improvement of 26% and 31% for these two techniques, respectively.

1 Introduction

Multithreading is one of the main techniques employed for tolerating latency [1, 2]. In multithreading, a thread suspends its execution whenever a long latency event is encountered. In such a case, the processor switches to another thread ready for execution. This form of multithreading is usually referred as blocking multithreading [2]. Another form of multithreading is non-blocking multithreading. In this case, a thread is scheduled for execution only if all of its input values are available in the local memory, thus no synchronization nor communication latencies will be experienced.

Data driven multithreading (DDM) is a non-blocking multithreading model of execution evolved from the dataflow model of computation [3]. An implementation of a data driven multithreaded architecture is the Data Driven Network of Workstations (D^{2}NOW) [4]. D^{2}NOW utilizes conventional control-flow workstations, augmented with an add-on card called the Thread Synchronization Unit (TSU). The TSU supports data driven scheduling of threads.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 561–570, 2004.
© Springer-Verlag Berlin Heidelberg 2004

A program in DDM is a collection of code blocks called the context blocks. A context block is equivalent to a function. Each context block comprises of several threads. A thread is a sequence of instructions equivalent to a basic block. A producer/consumer relationship exists among threads. In a typical program, a set of threads create data, the producers, which is used by other threads, the consumers. Scheduling of threads is done dynamically at run time by the TSU, based on data availability.

Data driven scheduling leads to irregular memory access patterns that affect negatively cache performance. This is due to the fact that threads are scheduled for execution based only on data availability without taking into account temporal or spatial locality. On the other hand, data driven scheduling allows for optimal cache management policies, by ensuring that the required data is prefetched into the cache, before a thread is fired for execution. Furthermore, we can ensure that data preloaded in the cache is not replaced before the corresponding thread is executed, thus reducing possible cache conflicts. We call this cache management policy the CacheFlow policy.

In this paper we examine three variations of the CacheFlow policy. In the first implementation we prefetch into the cache the data of the threads scheduled for execution in the near future. These threads are then placed in a firing queue and wait for their turn to be executed. We call this the *Basic Prefetch CacheFlow*. In the second implementation, called *CacheFlow with Conflict Avoidance*, we maintain a list of all addresses of the data prefetched for the threads in the firing queue, and make sure that this data is not evicted from the cache until the corresponding threads are executed. In the third implementation, called *CacheFlow with Thread Reordering*, we reorder the sequence of executable threads, before they enter the firing queue, in order to exploit spatial locality.

An execution driven simulator is used to evaluate the potential of the CacheFlow policy in reducing cache misses. The workload used on these experiments consisted of eight scientific applications, six of which belong to the Splash-2 suite [5]. Simulation results have shown a significant reduction in the cache miss ratio. This results in a speedup improvement ranging from 10% to 25% (average 18%) when the *Basic Prefetch CacheFlow* policy is used. A larger increase (14% to 34% with a 26% average) is observed when the *CacheFlow with Conflict Avoidance* is used. A further improvement (18% to 39% with a 31% average) is observed when the *CacheFlow with Thread Reordering* is employed.

2 Related Work

A variety of techniques such as data forwarding [6, 7], and prefetching [8, 9], have been proposed to tolerate the long memory access latency. With data forwarding a producer processor forwards data to the cache of consumer processors as soon as it generates it. The main drawback of data forwarding is that it may displace useful data from the consumer's cache. Our implementation of data driven multithreading model of execution employs the data forwarding concept in the sense that a producer node is responsible for forwarding remote data as

soon as it is produced to the consumer node, and that it employs only remote write operations. The difference in our approach is that data is forwarded to the consumer's main memory, not to the cache, avoiding the possibility of displacing useful data from the cache.

Data prefetching reduces cache misses by preloading data into the cache before it is accessed by the processor. A review on prefetching is presented by Vanderwiel and Lilja [10]. Data prefetching can be classified as hardware prefetching [11, 12], software prefetching [13, 14], or thread based prefetching [15–17]. Thread based prefetching is employed in multithreaded processors [1, 18] to execute a thread in another context that prefetches the data into the cache before it is accessed by the computation thread. CacheFlow employs compiler-assisted hardware prefetching mechanisms. The difference between CacheFlow and other hardware prefetchers is that most of the other prefetchers attempt to predict possible cache misses based on earlier misses, while in CacheFlow the addresses of the data needed by a thread scheduled for execution is either specified at compile time or it is determined at run time when the thread becomes ready for execution. CacheFlow has the advantages of both software and hardware prefetching. In addition, it avoids unnecessary prefetching that would lead to extra bus traffic and cache pollution.

3 The CacheFlow Policy

One of the main goals of Data Driven Multithreading is to tolerate latency by allowing the computation processor do useful work while a long latency event is in progress. This is achieved by scheduling threads based on data availability. An argument against data driven multithreading is that it does not fully exploit locality, since threads are scheduled for execution based only on data availability. Scheduling based on data availability, on the other hand, allows the implementation of efficient short-term optimal cache management. This paper focuses on the implementation of these policies which we named CacheFlow.

The implementation of the CacheFlow policy is directly related to the Thread Issue Unit (TIU), a unit within the TSU [4] responsible to schedule threads which are ready for execution. The other two units of the TSU are the Post Processing Unit (PPU) and the Network Interface Unit (NIU). Each thread is associated with a synchronization parameter, called the *Ready Count*. The PPU updates the *Ready Count* of the consumers of the completed threads, and determines which are ready for execution. The NIU is responsible for the communication between the TSU and the interconnection network. In this paper we present four implementations of the TIU which are described in the following sections.

3.1 TIU with No CacheFlow

The TIU without CacheFlow support is a simplified version of the TIU depicted in Figure 1. It consists only of the Waiting Queue (WQ), the Firing Queue (FQ) and the IFP part of the Graph Cache. The Graph Cache serves as a look-up

Fig. 1. The TIU that supports CacheFlow with Conflict Avoidance.

table. When a thread is deemed executable by the Post Processing Unit (PPU), its identification number (Thread#) and index are placed in the WQ. The TIU uses the thread identification number to determine the threads starting address by reading its IFP from the Graph Cache. After the thread's IFP is determined, the triplet Thread#, Index and IFP are shifted into the FQ and the thread waits for its turn to be executed. The last instructions in each thread read the starting address (IFP) of the next ready thread from the FQ and branch to that address.

3.2 TIU with Basic Prefetch CacheFlow Policy

To implement the CacheFlow policy with basic prefetch, two extra fields are added in the Graph Cache. These fields are determined at compile time and loaded in the Graph Cache at run time as Data Frame Pointer 1 (DFP1) and Data Frame Pointer 2 (DFP2). If a thread has only one input, then DFP1 contains the memory pointer to that value, while DFP2 is set to 0. If a thread has more than two inputs, then DFP1 is set to 0 while DFP2 is a pointer to the DFP list, a memory block within the TSU that contains a list of DFPs. Note that a thread input corresponds to an arrow in a dataflow graph. A thread input may be a single variable or a contiguous memory block that fits into one cache block.

When a thread becomes executable, i.e. all of its inputs have been produced, the Post Processing Unit (PPU) places the Thread# and index in the WQ. A thread is processed by first reading the Thread# and index from the WQ. The Thread# is used as a pointer to the Graph Cache that gives the IFP, DFP1 and DFP2 of the thread. The address of the data needed by the thread is obtained using the DFP with the index, thus the exact address of the data is determined dynamically, at run time. The Thread#, index and IFP are then shifted in the FQ and the thread waits for its turn to be executed. As soon as the addresses of the data needed by the thread are determined, they are sent to the Prefetch

Unit that snoops the processor to verify whether these addresses are already in the cache. If the required data is not in the cache, then a prefetch request is issued.

3.3 TIU with Conflict Avoidance (Optimization 1)

One disadvantage of the basic prefetch CacheFlow policy is that excessive traffic is placed on the processor's bus and snooping lines. Another disadvantage is that prefetching can cause cache conflicts, i.e. it is possible that a cache block required by a thread waiting in the FQ is replaced by another block, before the thread is executed. We call these conflicts *false cache conflicts* as they originate from the policy and not from the execution of the code. The possibility of false cache conflicts is reduced by keeping the size of the FQ as small as possible. A small FQ, on the other hand, increases the possibility that a thread is fired before its is prefetched. This becomes more critical for threads with a small number of instructions.

The TIU with *Conflict Avoidance* prevents the Prefetch Unit from replacing cache blocks required by the threads waiting in the FQ. This is achieved with the use of the Reserved Address Table (RAT) that contains the addresses of all cache blocks prefetched for the threads waiting in the FQ, as well as the thread currently running. All addresses required by a ready thread, removed from the WQ, are determined using the information from the Graph Cache, and placed in the Tag Queue. These addresses are then compared with the contents of the RAT to determine if prefetching would cause a cache conflict. A thread is shifted in the FQ if none of its addresses would result in a cache conflict. If it is detected that an address would result in a false cache conflict, then the tested thread is placed temporarily in a buffer, and the next thread from the WQ is tested. Threads waiting in the temporary buffer have precedence over the threads in the WQ. This is essential to avoid thread starvation, as a thread waiting in the temporary buffer is blocking its consumers from executing.

3.4 TIU with Thread Reordering (Optimization 2)

Both previous CacheFlow implementations address only the improvement of data locality. To exploit temporal code locality we have included a reordering mechanism that reorders the threads in the WQ. Threads with the same identification number are placed near each other in the WQ, increasing the probability that the code of a thread will be used many times before it is replaced from the cache. Furthermore, threads with the same identification number (Thread#), are ordered according to their index (iteration number), thus exploiting spatial data locality. Reordering reduces further snooping overheads on the processor and the bus. The thread reordering mechanism operates in parallel and asynchronously with the rest of the TIU and thus it does not add any extra delays in the datapath of the TIU.

The concept of thread reordering is depicted in Figure 2. Whenever a new thread becomes ready by the PPU, its Thread# is compared with the Thread#

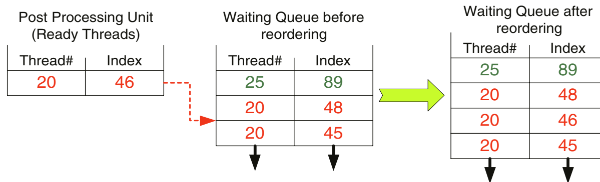

Fig. 2. Example of the WQ with the thread reorder Cache-Flow policy.

of the threads in the WQ. If a match is not found, the new thread is appended at the end of the WQ, otherwise the index of the new thread is compared with the index of the threads in the WQ with the same Thread#. The new thread is inserted in the WQ so that the index of the threads with the same Thread# appear in ascending order.

4 Evaluation Methodology

In order to evaluate the ability of the proposed cache management policy in reducing cache misses, we have built an execution driven simulator that uses native execution [19]. Both the host and the target processor is an Intel Pentium processor with a 256K L2 unified cache, and 16K L1 data and instruction caches. All caches are 4-way set associative with a 32-byte line size. Simulations were carried out for distributed shared virtual memory systems with 2, 4, 8, 16 and 32 processors. A clock cycle counter is maintained for each CPU, TSU unit and the interconnection network. The simulator uses the timings produced by the actual implementation of the TSU [4]. The time needed to execute each thread is obtained using the processor's time stamp performance counter [20]. Calls to functions that simulate the TSU and the interconnection network are interleaved with the execution of threads on the host processor, according to the clock cycle counter of each unit.

The cache miss rate is obtained using the processor's performance monitoring counters [20]. Since the machine used as the host is also the target, the state of the cache of the simulated application is affected by the simulation process. Therefore, the simulator performs extra operations to recover the system to the same state as it was before executing the simulation code.

Eight scientific applications are used to evaluate the three variations of the CacheFlow cache management policy. Six of these applications, *LU*, *FFT*, *Radix*, *Barnes*, *FFM* and *Cholesky* belong to the Splash-2 suite [5]. These applications have been modified to support data driven execution. The partition of the code into threads has been done manually. The creation of the data driven graph is done automatically by the simulator. The other two applications, *Mult* and *Trapez* represent standard algorithms used in large scientific applications such as the block matrix multiplication and the trapezoidal method of integration, respectively. To examine the effect of problem size on the effectiveness of the CacheFlow management we have used, for certain applications, two problem sizes: Data Size 1 (corresponding to 64K matrices) and 2 (corresponding to 1M matrices).

5 Results

5.1 Effect of Data Driven Sequencing on Miss Rate

Table 1 depicts the L2 cache miss rate for the sequential single threaded execution on a single processor and the different DDM configurations with CacheFlow on a 32-node system. Note that in order to avoid misleading results, for the measurement of the cache miss rate, we have scaled down the data size of the sequential single threaded execution to match the data size for each of the nodes in the 32-node DDM system, i.e. the data sizes used for the sequential single thread execution are the same as those used by each node in the data driven multithreaded execution. As expected, the baseline DDM configuration shows a higher miss rate than the sequential (increase from 7.1% to 9.8%), which corresponds to a 38% increase for the average of all applications. This reflects the loss of locality for both the code and data. The *Basic Prefetch CacheFlow* implementation reduces the miss rate from 9.8% to 3.2% (68% decrease compared to the baseline DDM). It is important to notice that the reduction achieved by the *Basic Prefetch CacheFlow* results in miss rate values lower than the original sequential execution. The use of the two CacheFlow optimizations results in further reductions on the miss rate, which becomes 1.9% and 1.4% respectively.

Table 1. Cache miss rate for Data Size 1.

Application	Miss Rate (Data Size 1 - DS1)				
	Sequential	DDM without Cache-Flow	Basic Prefetch (Buffer =16)	Optimization 1	Optimization 1 & 2
Mult	5.4%	7.5%	2.1%	1.4%	1.1%
Trapez	2.8%	3.6%	1.8%	1.3%	1.0%
LU	4.8%	6.1%	1.7%	0.9%	0.8%
FFT	7.5%	10.1%	3.0%	2.0%	1.2%
Barnes	7.4%	10.1%	2.6%	1.1%	0.8%
Radix	14.0%	18.1%	6.7%	4.1%	3.3%
FMM	8.3%	11.8%	3.2%	2.1%	1.6%
Cholesky	6.2%	11.3%	4.2%	2.4%	1.8%
Average Miss Rate	7.1%	9.8%	3.2%	1.9%	1.4%

5.2 Effect of Data Size on Miss Rate

The effect of problem size on the cache miss rate is presented in Table 2. By increasing the problem size by a factor of 16, the average cache miss rate for the sequential execution is increased from 7.9% to 9.7% (23% percentage increase). This increase is justified by the fact that the working set for the large problem size does not fit in the cache, resulting in more cache misses. The cache miss rate increase for the DDM execution without the CacheFlow management, is increased from 10.4% to 12.0% (16% percentage increase). The increase of the miss rate is significantly reduced when the different CacheFlow policies are used, (7%, 5% and 8% percentage increase respectively). This shows that CacheFlow is efficient in keeping the miss rate low independently of the problem size.

5.3 Effect of Firing Queue Size on Performance

Prefetching must be completed early enough to ensure that data is prefetched before the thread using that data is fired for execution. Nevertheless, prefetching must not be initiated too early, to avoid replacing cache blocks already prefetched by threads waiting in the Firing Queue (FQ). The effect, of the size of the FQ, for *Radix*, on the cache miss rate and the false conflicts when the *Basic Prefetch CacheFlow* is employed is depicted in Figure 3-(a). For these results a thread is shifted into the FQ as soon as the prefetching operation is initiated. The cache miss rate is higher when the FQ size is small. This is due to the fact that a thread might be fired before the prefetching is completed, resulting in cache misses. As the size of the FQ increases, there is more time for the prefetch unit to complete the prefetching operation, since the processor will execute other threads. Increasing the size of the FQ, increases also the number of false cache conflicts, resulting in more cache misses. The rest of the applications behave in a similar way. For all applications the minimum cache miss ratio is obtained when the FQ size is 16.

Table 2. Cache miss rate for Data Size 2.

Application	Miss Rate (Data Size 2 - DS2)				
	Sequential	DDM without Cache-Flow	Basic Prefetch (Buffer =16)	Optimization 1	Optimization 1 & 2
Mult	7.1%	9.2%	2.2%	1.5%	1.2%
LU	6.5%	7.4%	1.8%	1.0%	0.8%
FFT	9.1%	11.4%	3.1%	2.0%	1.3%
Radix	16.1%	19.8%	7.2%	4.3%	3.4%
Average Miss Rate	9.7%	12.0%	3.6%	2.2%	1.7%
Change (from DS1)	23%	16%	7%	5%	8%

To reduce the cache miss rate when the FQ is small, we have changed the FQ shifting policy. A thread is shifted only after prefetching is completed. This change affects also the CPU idle time. The effect, of the size of the Firing Queue (FQ), for *Radix*, on the CPU idle time when the *Basic Prefetch CacheFlow* is employed is depicted in Figure 3-(b). Measurements of the CPU idle time show that there is a significant increase in the CPU idle time when the FQ is too small. For all applications the CPU idle time when the FQ size is over 16 is the same as the idle time obtained without CacheFlow.

5.4 Effect of CacheFlow on Speedup

Figure 4 shows the effect of the three CacheFlow implementations on speedup, compared to sequential execution, for machine sizes ranging from 2 to 32 processors. A speedup improvement ranging form 10% to 25% (average 18%) is obtained when the it Basic Prefetch CacheFlow policy is used on a 32-processor system. A bigger increase (14% to 34% with a 26% average) is observed when the CacheFlow with *Conflict Avoidance* is used. A further improvement (18% to 39% with a 31% average) is observed when the *Thread Reordering* is employed.

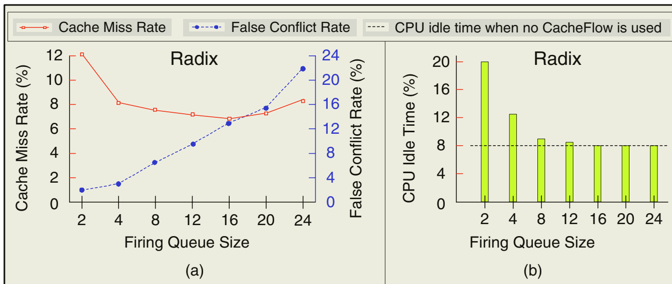

Fig. 3. FQ size effect on cache miss rate, false cache conflicts and CPU idle time.

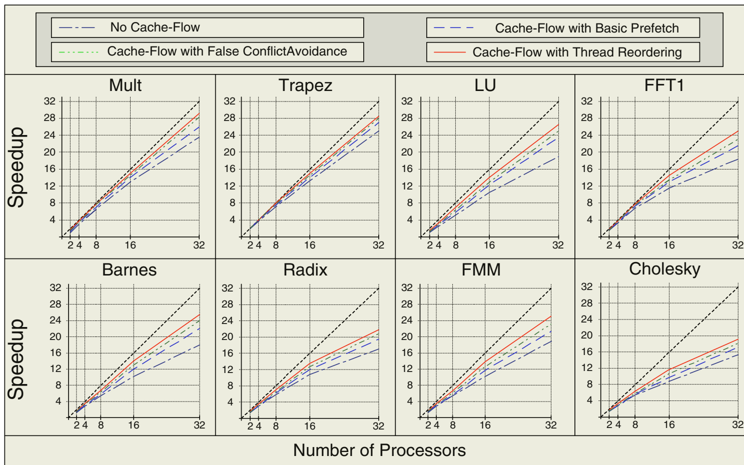

Fig. 4. Effect of CacheFlow and number of processors on speedup.

6 Conclusions and Future Work

Data Driven Multithreading is proposed as an execution model that can tolerate communication and synchronization latency. Nevertheless, data driven sequencing has a negative effect on performance due to loss of locality on the data access. In this paper we presented CacheFlow, a cache management policy that significantly reduces cache misses by employing prefetching. To avoid false cache conflicts and further exploit locality we proposed two optimizations: *Conflict Avoidance* and *Thread Reordering*.

Simulation results based on an execution driven simulator that runs directly on the host processor as well as measurements obtained from the developed hardware show that CacheFlow effectively reduces the miss rate. The basic prefetch implementation resulted in an average reduction in the cache miss rate of 67%, while the two optimizations resulted in further reductions: 81% and 86%, respectively. These reductions resulted in a speedup improvement on a 32-processor system of 18%, 26% and 31% respectively. An increase in the problem size by a

factor of 16 resulted in a very low increase in the cache miss ratio (7%, 5% and 8% respectively). Overall the results show that CacheFlow is an effective technique in tolerating memory latency, and an important enhancement for the data driven multithreading system. In the future we plan on extending the CacheFlow policy to implement it on SMT systems.

References

1. S. Eggers *et al.* Simultaneous multithreading: A platform for next generation processors. *IEEE Micro*, pages 12–18, September/October 1997.
2. J. Sile, *et al.* Asynchrony in Parallel Computing: From Dataflow to Multithreading. *Parallel and Distributed Computing Practices*, Vol.1, No.1, 1998.
3. P. Evripidou and J-L. Gaudiot. A Decoupled Graph/Computation Data-Driven Architecture with Variable Resolution Actors. In *In Proc. ICPP-1990*, August 1990.
4. P. Evripidou and C. Kyriacou. Data Driven Network of Workstations (D^2NOW), *Journal of Universal Computer Science (J.UCS)*, Volume 6/ Issue 10, Oct. 2000.
5. S. Cameron Woo *et al.* The SPLASH-2 Programs: Characterization and Methodological Considerations, *In Proc. ISCA-1995*, Jun. 1995.
6. D. A. Koufaty *et al.* Data Forwarding in Scalable Shared Memory Multiprocessors, *In IEEE Transactions on Parallel and Distributed Systems*, 1996.
7. D. A. Koufaty and J Torrellas. Compiler Support for Data Forwarding in Scalable Shared Memory Multiprocessors, *In Proc. ICPP-1999*, September 1999.
8. D. K. Poulsen and P. C. Yew. Data Prefetching and Data Forwarding in Shared Memory Multiprocessors, *In Proc. ICPP-1994*, August 1994.
9. P. Trancoso and J. Torrellas, The Impact of Speeding up Critical Sections with Data Prefetching and Data Forwarding, *In Proc. ICPP-1996*, 1996.
10. S. Vanderwiel and D. Lilja Data Prefetch Mechanisms, *ACM Computing Surveys*, Vol. 32, No. 2, June 2000.
11. T. Sherwood *et al.* Predictor-directed stream buffers, *In 33rd International Symposium on Microarchitecture*, Dec. 2000.
12. J. Collins *et al.* Pointer cache assisted prefetching, *In Proc. MICRO-35*, Nov. 2002.
13. T.C. Mowry *et al.* Design and evaluation of a compiler algorithm for prefetching, *In Proc. ASPLOS-V*, Oct. 1992.
14. C.K. Luk and T.C. Mowry, Compiler based prefetching for recursive data structures, *In Proc. ASPLOS-VII*, Oct. 1996.
15. J. Collins *et al.* Dynamic Speculative Precomputation, *In Proc. MICRO-34*, Dec. 2001.
16. A. Roth and G. Sohi. Speculative data-driven multithreading, *In Proc. HPCA-7*, Jan. 2001.
17. Y. Solihin *et al.* Using User-Level Memory Thread for Correlation Prefetching, *In Proc. ISCA-2002*, May 2002.
18. C.K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors, *In Proc. ISCA-2001*, June. 2001.
19. C. Kyriacou *et al.* DDM-SIM: An Execution Driven Simulator for Data Driven Multithreading. Technical report, University of Cyprus, 2004.
20. Intel Corp. IA-32 Intel Architecture: Software Developer's Manual. Volume 3: System Programming Guide, *Intel, 2003.*