Characterizing and Understanding the Bandwidth Behavior of Workloads on Multi-core Processors

Guoping Long, Dongrui Fan, and Junchao Zhang

Key Laboratory of Computer Systems and Architecture and
Institute of Computing Technology and Chinese Academy of Science
100080 Beijing and China
{longguoping,fandr,jczhang}@ict.ac.cn

Abstract. An important issue of current multi-core processors is the off-chip bandwidth sharing. Sharing is helpful to improve resource utilization and but more importantly and it may cause performance degradation due to contention. However and there is not enough research work on characterizing the workloads from bandwidth perspective. Moreover and the understanding of the impact of the bandwidth constraint on performance is still limited. In this paper and we propose the phase execution model and and evaluate the arithmetic to memory ratio (AMR) of each phase to characterize the bandwidth requirements of arbitrary programs. We apply the model to a set of SPEC benchmark programs and obtain two results. First and we propose a new taxonomy of workloads based on their bandwidth requirements. Second and we find that prefetching techniques are useful to improve system throughput of multi-core processors only when there is enough spare memory bandwidth.

Keywords: multi-core architecture, phase model, memory bandwidth.

1 Introduction

In recent years and multi-core architectures are explored extensively by chip architects to both exploit parallelism and better meet the power consumption envelope. Commercial processors with two to four cores are prevalent in industry. Many core designs have also been proposed and evaluated.

An important problem of current multi-core architectures is the off-chip memory bandwidth sharing. While bandwidth sharing improves resource utilization sometimes and it can also cause performance degradation of the running program. Moreover and due to the limitation of the pin count and the gap between the required memory bandwidth and which is proportional with the growth rate of the on-chip computation density and and the realistic off-chip memory bandwidth will become larger and larger as more and more cores are to be integrated on a single chip. However and to the best of our knowledge and there is not enough research work on characterizing the workloads from bandwidth perspective. Moreover and the understanding of the impact of the bandwidth constraint on performance is still limited.

In this work and we attempt to provide insight for two questions. One is how to characterize the bandwidth requirements of arbitrary programs. The other one is how

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 110–121, 2009.

to understand the impact of the shared memory bandwidth constraint on system performance. The answers to both questions are important for both architects and system software developers. First and chip architects need to know the bandwidth characteristics of the application in order to determine the amount of off-chip bandwidth and and the appropriate area ratio between the computation logic and on chip storage. Second and understanding bandwidth requirements of applications is also helpful for operating system developers to implement an optimum task scheduler and which can schedule tasks to achieve minimum bandwidth contention.

In this paper and we extend the concept of arithmetic to memory ratio (AMR) to arbitrary programs and and propose the phase execution model to characterize the bandwidth behavior of workloads. We partition the program execution into a series of phases and and then evaluate the AMR of each phase to study the bandwidth requirements. We make three contributions in this paper:

First and we propose an analysis methodology to characterize the bandwidth behavior of workloads. Second and we propose a new taxonomy of workloads based their bandwidth requirements. Third and we find that prefetching techniques are helpful to improve overall system throughput of multi-core processors only if there is enough spare memory bandwidth.

The rest of the paper is organized as follows. Section 2 presents the phase execution model. Section 3 applies the model to SPEC benchmark programs and discusses our findings. Section 4 discusses related research and the last section concludes the paper.

2 The Phase Model

2.1 The Abstract Processor Model

Before the phase model is defined, we assume an abstract multi-core processor model in order to make our problem scope clear. For each processing core, we model the on-chip memory hierarchy as a fully associative working set memory (WM), with the capacity of C data blocks. There are n processing cores connected together to share the same memory request FIFO queue (MRQ). The on chip inter-connection network could be in any reasonable topology (MESH, bus, crossbar, etc). It includes necessary arbitration logic to deal with situations when multiple cores need to issue requests to MRQ.

2.2 Definition of the Phase Model

We base the phase execution model on the memory trace[1], which contains all information required for understanding the memory system behavior, including the bandwidth requirements. Intuitively, we define a phase as a segment of the memory trace which accesses distinct data blocks. Starting from this simple intuition, we partition the whole memory trace of a program execution into a series of phases and evaluate the AMR of each phase. First, we establish the memory trace concept formally.

Definition 1. *An address element is a 3-tuple* $< T_s, T_e, addr >$*, where* T_s*,* T_e *and addr denote the time the request is issued, the time the request is acknowledged and the address (aligned to memory block boundary) of the request, respectively.*

We do not distinguish operation type, such as load or store, in the address element. But we do care about whether an address hits in WM or not. If the *addr* field of an address element hits in WM, then $T_s = T_e$. Otherwise an off-chip memory request is generated for *addr*, and $(T_e - T_s)$ denotes the off-chip memory latency of the request. In this paper, we only take the off-chip latency into account, because only off-chip memory requests contribute to the memory bandwidth consumption.

Definition 2. *A memory trace is a ordered set A of N address elements:* $A = \{A_1, A_2, A_3, ..., A_N\}$.

We do not model multiple outstanding memory requests simultaneously in this paper. In other words, each element must issue after its previous one has finished. Therefore, $\forall A_k (1 \leq k \leq N)$, we have $A_{k+1}[T_s] > A_k[T_e]$[1]. Based on the memory trace concept, we can formally define the phase concept as follows.

Definition 3. *An ordered partition P of a memory trace* $A = \{A_1, A_2, A_3, ..., A_N\}$ *is an ordered set of* $m + 1 (m \geq 0)$ *ordered sets* $P = \{P_1, P_2, P_3, ..., P_{m+1}\}$, *where* $1 \leq k1 < k2 < ... < km \leq N$, $P_1 = \{A_1, A_2, ..., A_{k1}\}$, $P_2 = \{A_{k1+1}, A_{k1+2}, ..., A_{k2}\}$, ..., $P_{m+1} = \{A_{km}, A_{km+1}, ..., A_N\}$.

For example, one ordered partition of the memory trace $A = \{A_1, A_2, A_3, A_4, A_5\}$ is: $P = \{\{A_1, A_2\}, \{A_3\}, \{A_4, A_5\}\}$.

Definition 4. *The reference set of an ordered set of K address elements* $S = \{A_1, A_2, ..., A_K\}$ *is a set of memory addresses F(A) and is defined by the following formula:* $F(S) = \bigcup_{i=1}^{K} \{A_i[addr]\}$.

Note that the reference set of an ordered set A is not an ordered set. We can now define the notion of the phase trace and execution phases precisely.

Definition 5. *A phase trace is an ordered partition* $PT = \{P_1, P_2, ..., P_m\}$ *of a memory trace* $A = \{A_1, A_2, A_3, ..., A_N\}$, *with the following two conditions both satisfied*[2]*: (1)* $|F(P_1)| = C (C > 0)$*; (2)* $\forall k (1 < k \leq m), |F(P_k - P_{k-1})| = C$

Definition 6. *Given a phase trace* $PT = \{P_1, P_2, ..., P_m\}$, *each sub-partition* $P_i (1 \leq i \leq m)$ *is called an execution phase of the phase trace.*

Given an execution phase, we are interested in the ratio between the arithmetic computation time and the off-chip memory latency within the phase. As will be seen in the next section, it has strong impact on the performance when multiple cores shared the memory bandwidth. Precisely, we define arithmetic to memory ratio (AMR) as follows:

Definition 7. *The arithmetic to memory ratio (AMR) of a phase* $PH = \{E_1, E_2, ..., E_p\}$ *is defined by the following formula:*

[1] In this paper, we use [ ] operator to reference an address element's fields.

[2] Let A and B be two sets. In this paper, $|A|$ denotes the cardinality of set A. And $(A - B)$ denotes the set of all elements which belong to A but not B.

$$AMR_{PH} = \frac{E_p[T_e] - E_1[T_s] - \sum_{i=1}^{p} (E_i[T_e] - E_i[T_s])}{\sum_{i=1}^{p} (E_i[T_e] - E_i[T_s])} \quad (1)$$

Empirically, the larger the WM capacity is, the less the demand with the memory bandwidth. For the same memory trace, the phase trace is dependant on the choice of WM size. The larger WM is, the shorter the phase trace. The phase trace notion captures the intuition which exists in real processors. With the phase trace, we are particularly interested in the AMR of each phase, which reveals runtime program locality and bandwidth requirements.

2.3 Bandwidth Sharing Results

In this section, we focus on a special class of programs, for which all phases of the execution have the same AMR. Now supposing processing cores each executing an instance of such a program, we obtain two interesting results regarding bandwidth sharing. The first result targets such a question: Under what condition the shared bandwidth will be a performance bottleneck for multi-core processors? The second result targets another problem: Although it is well known that prefetching can help to improve performance, but can it always help to improve the overall system throughput on multi-core processors?

While quantitatively studying both problems for arbitrary programs on real systems is prohibitively complex, we give a precise result for a simplified version of the problem based on the phase model. Although simplified, we believe these results have interesting implications and can help to promote understanding of the bandwidth constraint of commercial multi-core processors on real workloads.

Theorem 1. *Suppose n cores share the MRQ and each core executes an instance of the program with uniform arithmetic to memory ratio: PAMR.Then iff* $n \leq \lfloor PAMR \rfloor + 1$*, there exists a memory request schedule in MRQ that no core suffers from performance loss due to bandwidth sharing.*

The proof is simple and thus is omitted due to page constraint. The main idea is that the data fetching time of each phase can be overlapped with the phase execution time of other phases. A possible schedule is shown in Figure 1(left), in which PF and PE denote the data fetching and execution of the phase, respectively. Theorem 1 demonstrates that the bandwidth requirement is closely related to the AMR of execution phases.

Commercial multi-core processors usually have prefetching capability, which also consumes the shared bandwidth. Based on the phase model, Theorem 2 gives the quantitative relationship between the AMR and overall system throughput on multi-core processors.

Theorem 2. *Suppose n processing cores with prefetching capability share the MRQ and each core executes an instance of the program with uniform arithmetic to memory ratio: PAMR. Then phase preteching can help to improve overall system throughput iff* $n \leq \lfloor PAMR \rfloor$.

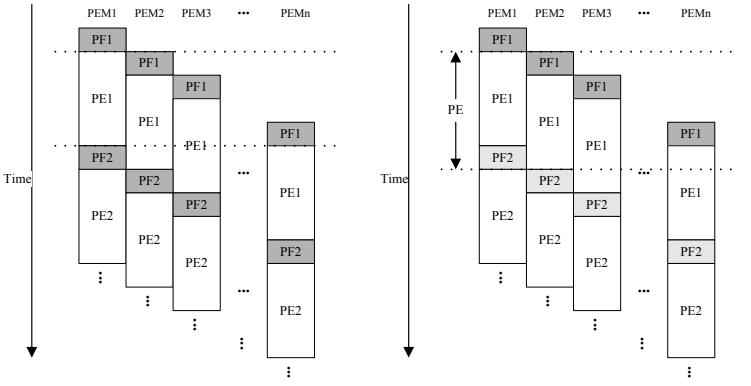


Fig. 1. A schedule without prefetching (left) or with prefeching(right)

We omit the proof here as well due to page constraint. When we say a core has prefetching capability, we mean it can prefetch data for the next phase during the execution of the current phase, and we assume the prefetch buffer is large enough to hold all prefetched data. Figure 1(right) shows the schedule with prefetching which illustrates the theorem. The most important implication of Theorem 2 is that, prefetching can help to improve overall system throughput only if there is enough spare memory bandwidth. Note that although in uniprocessor systems there is usually spare bandwidth for data prefetching, however, this is not always true for multi-core processors in which all cores share the bandwidth.

3 Application of the Phase Model

3.1 Experimental Methodology

In this section, we apply the phase model to SPEC benchmark programs. Since the model is based on the memory trace, we need a processor pipeline model to generate accurate traces. The simulation environment which we use to collect traces is the Godson-T simulator [2,3,4], which is designed for many-core simulation. The processing core has an in order 9-stage instruction pipeline and implements MIPS instruction set. Since the memory trace strongly depends on the processor architecture, please refer to [3,4] for more description of the details. The benchmark programs are given in Table 1. We run each program for ten billion instructions. All memory traces are analyzed assuming the WM size of 1MB. We conduct two sets of analysis based on the memory trace obtained from the simulator. The first one is the bandwidth behavior analysis, which is done by calculating the AMR distribution of each program. The other one is the performance impact analysis of shared bandwidth. We develop a phase analysis tool to predict the performance loss of each program. The inputs to the tool are phase traces generated from the memory traces of different programs. And the outputs are performance degradation results for all programs due to bandwidth sharing. The shared bandwidth is modeled by a FIFO queue. The MRQ can accept one phase fetch request a time from one phase trace. All requests in MRQ are processed in strict FIFO order.

Table 1. Benchmark Classification

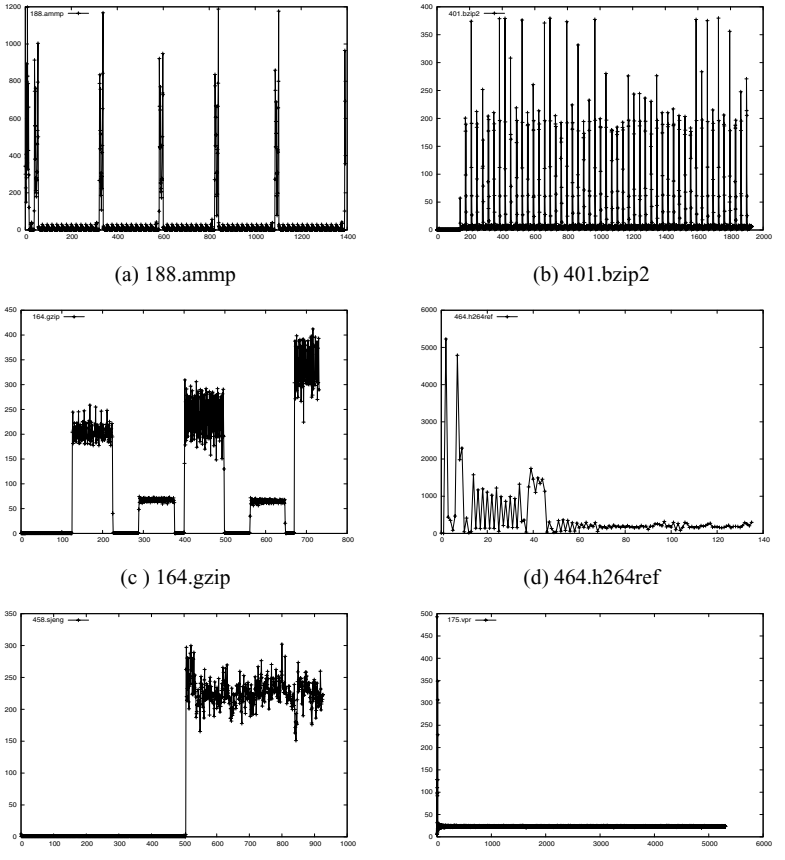
Bandwidth requirements	Program	# phases	AVG. AMR
Low	164.gzip	732	102.8
	458.sjeng	927	103.2
	464.h264ref	136	433.1
Medium	188.ammp	1393	44.4
	401.bzip2	1931	24.2
	175.vpr	5304	23.8
	300.twolf	6657	22.9
High	179.art	66389	0.67
	183.equake	9910	5.74
	470.lbm	64911	1.37
	433.milc	18443	2.37
	482.sphinx3	19183	4.12
	181.mcf	8438	3.11

In this paper, a memory block refers to the data transfer unit between the processor and memory. In cache based systems, a memory block is a last level cache line. In some processors [5], the front side bus supports burst data transfer of multiple words within a memory block. The analysis tool abstracts all implementation details of memory system away and model it as a bandwidth parameter B, which denotes the latency required to bring a memory block from off-chip memory. Suppose a phase fetch request needs to bring C blocks of data to working set memory, the latency introduced by a phase fetch request without bandwidth contention is $B * C$.

3.2 Bandwidth Behavior Analysis

The third and fourth columns of Table 1 show the number of phases observed and the average AMR of each program, respectively. Based on average AMR, we can classify all programs into three categories. Six programs (179.art, 183.equake, 181.mcf, 470.lbm, 433.milc and 482.sphinx3) have very low average AMR, indicating high off-chip bandwidth requirements. Three programs (164.gzip, 458.sjeng and 464.h264ref) have relatively small number of phases and very high average AMR, indicating very low bandwidth requirements. There are also programs (188.ammp, 401.bzip2, 175.vpr and 300.twolf) with medium average AMR, indicating medium demand of bandwidth.

Figure 2(a) to (f) show the AMR distribution for six programs. The x-axis denotes execution time, which is measured as the number of phases executed. The y-axis denotes the AMR of each phase. For the other seven programs, 300.twolf, 179.art, 470.lbm, 181.mcf, 433.milc and 482.sphinx have similar histograms with 175.vpr, and 183.equake has similar histogram with 401.bzip2. We omit the AMR histograms of these programs due to page constraint. As can be seen, different programs have dramatically different AMR distributions. Even for the same program (188.ammp, 164.gzip, 458.sjeng, etc), AMR distributions at different execution periods are also different. We can also divide all thirteen programs into three categories according to the AMR distribution: (1) Programs with periodic execution behavior, including 164.gzip, 188.ammp, 401.bzip2 and 458.sjeng. (2)



(a) 188.ammp (b) 401.bzip2

(c) 164.gzip (d) 464.h264ref

(e) 458.sjeng (f) 175.vpr

Fig. 2. AMR distribution of program phases

Programs with uniform behavior, including 175.vpr and all other 7 programs which are not shown. (3) Programs with irregular behavior, including 464.h264ref.

3.3 Bandwidth Sharing of Multiple Program Instances

Figure 3 shows results for multiple instances of the same program sharing the off chip memory bandwidth, assuming all instances are started at the same time. The x-axis denotes the number of cores, and the y-axis denotes the normalized performance of the worst core due to bandwidth sharing.

The average AMR has important impact on bandwidth contention. Some programs, including 183.equake, 482.sphinx3, 181.mcf, 433.milc, 470.lbm and 179.art, exhibit poor bandwidth sharing behaviors. This can be partially explained by their low average AMRs (¡6). For programs with relatively higher average AMRs (¿20), including 175.vpr, 300.twolf, 401.bzip2 and 188.ammp, the performance starts to drop when the number of cores increases beyond 16. The results for 464.h264ref, 164.gzip and

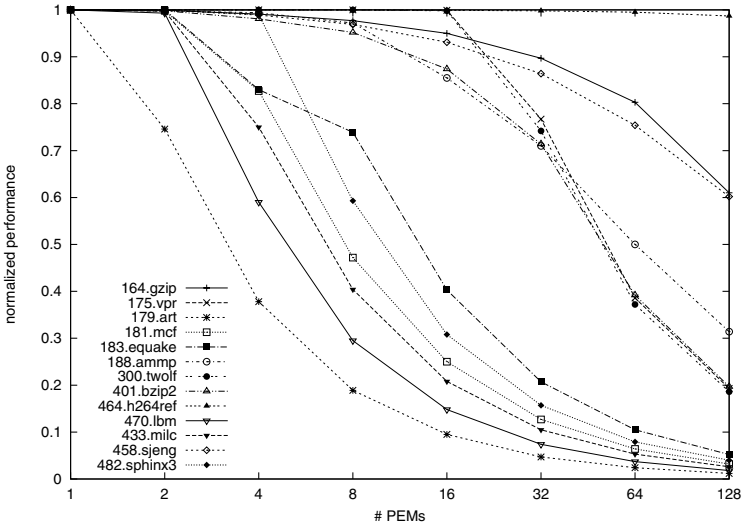


Fig. 3. Bandwidth sharing of multiple program instances

458.sjeng are much better than other programs. Although 164.gzip and 458.sjeng have very large average AMR (¿100), performance gets worse when PEM number is larger than 64 mainly due to the periodic nature of their phase behavior.

For programs (175.vpr and 300.twolf) with uniform phase behaviors, the performance loss due to bandwidth contention only occurs when core number is larger than the average AMR, as demonstrated in Theorem 1. However, for programs (164.gzip and 458.sjeng) with periodic behavior, the contention overhead slows down the performance when core number (64) is much less than the average AMR (> 100).

3.4 Bandwidth Sharing of Multiple Programs

Bandwidth contention of multiple programs is more complex than that of multiple instances of the same program. Table 2 shows normalized performance data (normalized to performance without bandwidth contention) when any two of the programs share the MRQ. For example, if 179.art and 181.mcf run together, the normalized performances of 179.art and 181.mcf are 0.95 and 0.91, respectively. That is, the performance overheads due to bandwidth sharing for 179.art and 181.mcf are 5% and 9%, respectively.

There are two observations to note in Table 2. First, for programs with high average AMR, bandwidth contention is not a problem because only two programs share the bandwidth. Second, if a program P1 with low average AMR and another program P2 with relatively high average AMR run together, the performance loss due to contention for P2 is worse than P1. For example, bandwidth sharing can cause higher performance loss to 181.mcf, 183.equake, 300.twolf, 470.lbm, 433.milc, and 482.sphinx3 than to 179.art. Operating systems with runtime AMR information can avoid scheduling programs with poor locality on cores of the same processor.

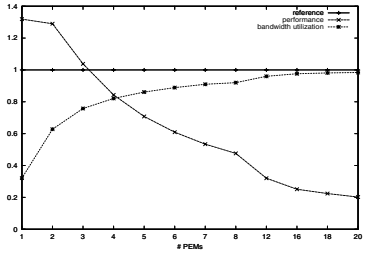
Table 2. Performance Results of Two Programs Sharing the Bandwidth

	gzip	Vpr	Art	Mcf	eqk	Amp	twlf	Bzip	h264	Lbm	milc	seng	sphx
Gzip	1.0	1.0	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Vpr	1.0	1.0	0.95	0.97	1.0	1.0	1.0	1.0	1.0	0.99	0.98	1.0	0.99
Art	1.0	1.0	0.74	0.95	0.98	0.99	0.99	0.99	1.0	0.91	0.97	1.0	0.96
Mcf	1.0	1.0	0.91	0.99	0.99	0.99	1.0	0.99	1.0	0.99	0.97	1.0	0.99
Eqk	1.0	1.0	0.88	0.94	1.0	1.0	1.0	0.99	1.0	0.96	0.97	1.0	0.98
Amp	1.0	1.0	0.98	0.99	0.99	0.99	1.0	1.0	1.0	0.99	0.99	1.0	0.99
Twlf	1.0	1.0	0.94	0.98	0.99	0.99	1.0	1.0	1.0	0.98	0.99	1.0	0.99
Bzip	1.0	1.0	0.98	0.99	0.99	1.0	1.0	0.99	1.0	0.99	0.99	0.99	0.99
H264	1.0	1.0	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Lbm	1.0	1.0	0.74	0.94	0.99	1.0	0.99	0.99	1.0	0.99	0.95	0.99	0.98
Milc	1.0	0.99	0.85	0.92	0.98	0.99	0.99	0.99	1.0	0.94	0.99	1.0	0.96
Seng	1.0	1.0	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
sphx	1.0	1.0	0.87	0.95	0.99	0.99	1.0	0.99	1.0	0.94	0.97	1.0	1.0

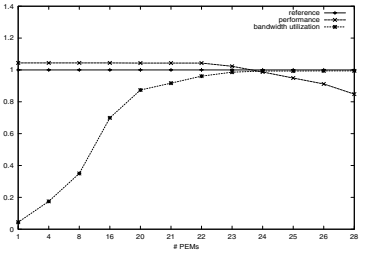
3.5 Prefetching Effects

Figure 4 shows results for multiple program instances sharing the bandwidth with phase prefetching. The x-axis denotes the core number. For the reference curve and performance curve shown in the figure, the y-axis denotes the performance normalized to execution without bandwidth contention and phase prefetching. And for the bandwidth utilization curve, the y-axis denotes the utilization of memory bandwidth. Only two representative programs, 181.mcf(Figure 4(a)) and 300.twolf (Figure 4(b)) are shown due to page limits. Note that the methodology can be applied to all other programs.

As can be seen, phase prefetching is a very effective technique to hide memory latency for 181.mcf. Without bandwidth contention, performance improvement reaches more than 30%. However, performance drops quickly as the core number becomes larger than 3, even if there is spare memory bandwidth. For 181.mcf, phase prefetching is helpful only if bandwidth contention is very small. For 300.twolf, phase prefetching can only bring limited performance improvement (4%) because the average AMR is relatively large (22.9). In this case, normalized performance drops below 1 only when



(a) 181.mcf



(b) 300.twolf

Fig. 4. Bandwidth sharing of multiple program instances

Table 3. Phase Model Accuracy

Name	2 Cores	4 Cores	8 Cores	16 Cores	32 Cores	64 Cores
164.gzip	0.20%	0.10%	0.10%	0.00%	0.00%	0.00%
175.vpr	0.00%	0.00%	0.00%	0.00%	4.70%	2.20%
181.mcf	4.40%	3.50%	0.40%	0.10%	0.10%	0.10%
300.twolf	0.00%	0.00%	0.00%	0.10%	2.80%	2.30%
179.art	1.30%	0.00%	0.00%	0.00%	0.00%	0.00%
183.equake	3.60%	2.90%	1.30%	0.20%	0.10%	0.10%
188.ammp	0.10%	0.10%	0.20%	0.20%	0.10%	0.40%
401.bzip2	0.20%	0.20%	0.20%	0.30%	3.30%	2.10%
458.sjeng	0	0.00%	0.00%	0.00%	0.00%	0.00%
464.h264ref	0.10%	0.00%	0.00%	0.00%	0.00%	0.00%
470.lbm	4.50%	3.20%	1.20%	0.60%	0.30%	0.00%
433.milc	4.90%	1.30%	0.10%	0.10%	0.10%	0.10%
482.sphinx3	0.60%	1.70%	2.60%	0.00%	0.00%	0.00%

the core number is larger than 23, for two reasons: (1) The bandwidth usage of 300.twolf saturates (> 99%). As demonstrated in Theorem 2, prefetching is useful only when there is spare memory bandwidth; (2) Bandwidth contention also results in performance loss.

3.6 Phase Model Validation

Previous experimental results are based on the phase analysis tool. To validate the accuracy of the analysis, we compare the predicted results obtained from the tool (shown in Figure 3) against that obtained with cycle accurate execution of these programs on Godson-T many-core simulator. Table 3 shows the results for each program. Start from the second column, each one shows the performance variation when multiple instances of the same program are used as input to the tool. For example, the 8Core column shows the performance variation between the performance predicted by the tool and the cycle accurate execution performance when eight instances are running simultaneously. As can be seen from the table, the maximum performance variation for all programs is within 4.9%. Therefore, the phase model is reasonably accurate for the basic performance trend analysis of bandwidth sharing. Note that although we only present results for multiple instances of the same program, analysis of the sharing behavior of different programs can be done easily as well.

4 Related Work

Bandwidth sharing is a critical issue for multi-core processors. Previous works [6,7,8] have been done to understand the bandwidth sharing in multi-core processors. In this work, we take a different approach by generalizing the arithmetic to memory ratio to general applications, and report our results on the impact of bandwidth sharing on performance.

Characterizing programs with arithmetic to memory ratio (AMR) has previously appeared in works on stream or many-core processor designs [9], for which programs with

regular arithmetic structure and high computation density can get high performance. In this paper, we extend the AMR concept to characterize the behavior of arbitrary programs. We show that the AMR distribution of execution phases has important connection with the bandwidth demands.

The idea of execution phase in this paper is partially inspired by previous works on performance optimizations [2,10,11]. [2,10] proposed optimization techniques for IBM C64 architecture, the idea is to partition the program into multiple phases, and overlap the computation and data transfer between the processor and memory explicitly. [11] proposed a new prefetching technique, called epoch model, to improve the processor performance. In this paper, we define the phase model formally above the memory trace. And study the AMR distribution among all execution phases to understand the bandwidth requirements of the program on multi-core architectures.

5 Conclusion

Bandwidth sharing is an important problem for current multi-core processors. In this paper, starting from the memory address trace, we formally define the phase model. The AMR distribution among all phases in the phase trace provides an interesting window to understand the bandwidth behavior of arbitrary programs. Based on the phase trace, we develop a trace driven bandwidth analysis tool to quantitatively study the impact of shared memory bandwidth constraint on performance. More importantly, we propose a new angle to classify the workloads based on bandwidth requirements.

There are two important conclusions from our experimental results. First, the bandwidth requirement of a program depends on the AMR distribution of its phase trace. Programs with high average AMR are more sensitive to bandwidth sharing than those have low average AMR. Second, in multi-core processors with shared bandwidth, prefetching techniques are useful to improve overall throughput only when there is spare memory bandwidth.

Different application areas may have dramatically different demands on off-chip memory bandwidth. For chip architects, the phase model is helpful to understand the bandwidth behavior of applications. In future work, we will investigate the possibility of passing runtime bandwidth information to OS kernels. With this information, the OS kernel can schedule tasks with minimum bandwidth contention to multiple cores of the same processor.

Acknowledgements

This work is supported by the national grand fundamental research 973 program of China under grant No. 2005CB321600, the national high-tech research and development (863) plan of China under grant No. 2009AA01Z103, the national natural science foundation of China under grant No. 60736012, the EU MULTICUBE project under grant No. FP7-216693, and Beijing natural science foundation under grant No. 4092044. In particular, the authors would like to thank anonymous reviewers for their constructive comments to this paper.

References

1. Uhlig, R., Mudge, T.: Trace-driven memory simulation: A suvey. ACM Computing Surveys 29(2) (June 1997)
2. Tan, G.M., Fan, D.R., Zhang, J.C., Russo, A., Gao, G.R.: Experience on optimizing irregular computation for memory hierarchy in manycore architecture. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (February 2008)
3. Yuan, N., Yu, L., Fan, D.: An efficient and flexible task management for many-core architecture. In: Proceedings of Workshop on Software and Hardware Challenges of Manycore Platforms, In conjunction with the 35th International Symposium on Computer Architecture (June 2008)
4. Long, G.P., Fan, D.R., Zhang, J.C.: Architectural support for cilk computations on many-core architectures. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (February 2009)
5. Hu, W.W., Zhang, F.X., Li, Z.S.: Microarchitecture and performance of godson-2 processor. Journal of Computer Science and Technology 20(2) (2005)
6. Rob, A.P., Mandal, F.A., Lim, M.Y.: Empirical evaluation of multi-core memory concurrency initial version (January 2009)
7. Weidendorfer, J.: Understanding memory access bottlenecks on multicore (2007)
8. Ahsan, B., Zahran, M.: Cache performance, system performance, and off-chip bandwidth... pick any two. In: Proceedings of INA-OCMC (2009)
9. Long, G.P., Fan, D.R., Zhang, J.C., Song, F.L., Yuan, N., Lin, W.: A performance model of dense matrix operations on many-core architectures. In: Proceedings of European Conference on Parallel and Distributed Computing (August 2008)
10. Tan, G.M., Sun, N.H., Gao, G.R.: A parallel dynamic programming algorithm on a multi-core architecture. In: Proceedings of the Annual ACM Symposium on Parallelism in Algorithms and Architectures (2007)
11. Chou, Y.: Low-cost epoch-based correlation prefetching for commercial applications. In: Proceedings of International Symposium on Microarchitecture (2007)