A Study on Performance Benefits of Core Morphing in an Asymmetric Multicore Processor

Anup Das, Rance Rodrigues, Israel Koren and Sandip Kundu
Department of Electrical and Computer Engineering
University of Massachusetts at Amherst
Email: {anupdas, rodrigues, koren, kundu}@ecs.umass.edu

Abstract—**Multicore architectures are designed so as to provide an acceptable level of performance per unit power for the majority of applications. Consequently, we must occasionally expect applications that could have benefited from a more powerful core in terms of either lower execution time and/or lower energy consumed. Fusing some of the resources of two (or more) cores to configure a more powerful core for such instances is a natural approach to deal with those few applications that have very high performance demands. However, a recent study has shown that fusing homogeneous cores is unlikely to benefit applications. In this paper we study the potential performance benefits of core morphing in a heterogeneous multicore processor that can be reconfigured at runtime. We consider as an example a dual core processor with one of the two cores being designed to target integer intensive applications while the other is better suited to floating-point intensive applications. These two cores can be fused into a single powerful core when an application that can benefit from such fusion is executing. We first discuss the design principles of the two individual cores so that the majority of the benchmarks that we consider execute in a satisfactory way. We then show that a small subset of the considered applications can greatly benefit from core morphing even in the case where two applications that could have been executed in parallel on the two cores are run, for some percentage of time, on the single morphed core. Our results indicate that a performance gain of up to 100% is achievable at a small hardware overhead of less than 1%.**

I. Introduction

Technology scaling has permitted an ever increasing number of transistors to be integrated per unit area, while running at higher frequencies. This had lead to a tremendous increase in power density. When power density became a pressing issue and multi-gigahertz frequencies could not be sustained, processor manufacturers responded with lower frequency integrated processors featuring multiple cores. When multiple cores are integrated on a die, they are still limited by an overall power dissipation envelope that stems from packaging and cooling technologies. Consequently, most chip multiprocessor (CMP) systems integrate cores of relatively moderate capability as integration of extremely capable cores will result in higher cost and breaching of dissipation limits. This is also consistent with the typical mix of workloads run by a processor. For the majority of applications, the capability of cores found in today‘s CMP system is quite acceptable. However, we must occasionally expect applications that could have benefited from a more powerful core in terms of either lower execution time and/or lower energy consumed. Fusing the resources of two cores to configure a larger core seems to be a natural solution. However, researchers have found that most applications have limited Instruction Level Parallelism (ILP) [17]. Therefore, integrating resources from several cores to increase parallelism in execution is unlikely to result in improved performance, unless the underlying execution engines become more powerful themselves. As we have indicated, it is infeasible for every core to have high performance execution engines due to area and power constraints. To overcome this problem, we propose to use asymmetric CMPs.

CMP cores may be symmetric (SCMP) or asymmetric (ACMP). It is well known that different workloads require different processor resources for high performance. Some workloads are load-store intensive, some are integer ALU intensive, while others are floating-point (FP) intensive or memory bus intensive or any combinations of the above. Also, the resource requirements of these workloads may vary with time due to changes in program phases [3]. Accordingly, we propose an ACMP architecture where different cores have strength in different areas. There are several benefits of this approach. First, it allows certain programs or program phases to run on a core at a very high level of performance. Second, each core remains modest in size, which allows the ACMP to meet cost and power targets. Third, when operated as an uniprocessor, the combined processor acquires strength in all areas of execution allowing much higher performance for those programs that benefit from a higher level of processor parallelism. The disadvantage of this approach is that workloads are naturally affine to certain cores for performance. To take advantage of ACMP, this thread to core affinity must be discovered and the tasks must be scheduled accordingly. This problem has been addressed previously in multiple contexts [3-4]. We assume that such a solution exists to take advantage of an ACMP system that can also morph itself to run as a uniprocessor.

The processor parameters are decided upon by experimentation, so that satisfactory performance is achieved for most workloads and hence cannot take advantage of situations where a workload with different characteristics than those tested is encountered. Under such conditions, a reconfiguration of the CMP on the fly to match the demands of any incoming workload is an attractive approach. By contrast, reconfiguring an SCMP as a uniprocessor offers more of the same resources and hence its performance saturates as the ILP saturates.

Further, SCMPs are limited to optimizing a single core designed so that high performance is achieved for a large set of workloads [9]. This can be a problem if sequential as well as parallel performance is required, as the design principles for achieving these two goals are different [2],[6], hence a compromise must be arrived at in the design. A reconfigurable ACMP offers a more appealing alternative as it can provide both strong performance and less idling. In an ACMP, a few cores can be designed for strong FP performance and the rest for ALU performance. Thus, unlike SCMP, when a workload is matched against processor resources, an ACMP has a greater potential to deliver performance while idling fewer resources. For the purpose of our experiments, the analyzed ACMP consists of two cores, one designed to perform best for ALU intensive and the other for FP intensive applications. Therefore, both ALU and FP intensive applications can be handled with ease. The main feature of our paper is a core morphing technique, where depending upon the current workload, an automatic reconfiguration of the cores in the ACMP into a strong uni-processor takes place to best suit the needs of the workload. The design parameters used for each of the cores and the justification of choices is explained in the later sections of the paper. The experimental results demonstrate the performance advantage of the proposed system. Our results indicate that despite some hardware overhead for control and communication to support core morphing, the performance gains far outweigh the limited hardware overhead for many applications.

The rest of the paper is organized as follows: In Section II, we survey the prior research on this topic and show the key differences between our approach and those previously proposed. In Section III, the hardware design is described. In Section IV, the experiments to determine the base architecture are described. Results and analysis are presented in section V. Section VI presents the conclusions.

II. Previous work

The recent literature on reconfigurable processor fabrics can be broadly classified into SCMPs and ACMPs. In this section we provide a brief overview of the proposed ideas and distinguish between those and our proposed architecture. CMPs have recently received considerable attention with the increasing levels of ILP in applications. Core fusion [8] involves the use of a homogeneous CMP, reconfigurable at runtime into stronger cores by fusing resources from the available cores. Another approach to fusion of homogeneous cores is presented in [10], where 32 dual-issue cores can be fused into a single 64-issue processor. However, both methods suffer from inter-core communication overheads. Also, the reconfiguration overhead of critical units like the ROB, issue queue and load/store queue, affects the ability to maximize benefits. The hurdle in achieving good performance by fusing simple in-order cores into out-of-order (OOO) cores has been shown in [2]. In [11], the authors propose Conjoined core architecture to achieve a compromise between Simultaneous Multithreading (SMT) and CMPs. They employ time sharing between resources and report the tradeoffs between area savings and performance slowdown. Heterogeneous architectures to achieve power/performance benefits have also been explored. In [3], Kumar et al. show the benefits in terms of power reduction using a single ISA heterogeneous CMP. They map different applications to different cores using criteria such as reduced power consumption. Energy related benefits have also been explored in [1], where single ISA heterogeneous cores working at different frequencies are used. In [6], the authors explore various methods of reducing power consumption and report that heterogeneous cores are the most useful in such situations. Benefits of heterogeneous architectures for performance gains with multi-programmed workloads have been explored in [4]. Higher performance per area and performance per watt have been shown for asymmetric cluster CMP architectures in [12]. All the above mentioned publications consider off the shelf cores and do not design them from scratch. In [13], the authors address the design of an ACMP such that area and power efficiency are achieved. They use cores that match the resource requirements of certain types of workloads. They also report performance gains of up to 40%.

III. Our Approach

Our proposed morphing architecture is distinct from prior work in multiple regards. First, we use a dual core ACMP, with one core designed for strong ALU performance (called strongALU core) while the other is designed for strong FP performance (called strongFPU core). Second, we keep the design and communication overhead low by using the front end of just one processor instead of fusing the fetch/decode/issue units. Third, we focus on a fusing just 2 cores, to reduce the complexity of backend execution units. Fourthly, we turn-off the weak execution units, such as the ALU of the strongFPU core and the FPU of the strongALU core to keep the issue logic simple.

In the normal mode of operation, the cores work independently of one another. This mode works well for applications that are parallel. However, whenever an application that requires high single threaded performance is encountered, a dynamic reconfiguration of the two cores into a single core takes place. There are several automatic approaches for detecting such an event [8],[10]. In this paper we assume that changes in the Instructions Per Cycle (IPC) metric can trigger an ACMP to uni-processor or a uni-processor to ACMP reconfiguration.

Upon morphing into a uni-processor, the strongALU core retains its frontend (fetch/decode, ROB) and ALU units while taking control of the FPU units from the strongFPU core. In the proposed design, the strongALU core has a better front end and weak FPU units, while the strongFPU core has a weaker front end but a valiant FPU unit. Hence, by taking charge of the FPU units from the StrongFPU core, the morphed core is formed with a strong front end and strong ALU and FPU units. This is depicted in Fig. 1. As a result, the morphed core is capable of handling applications that demand high single threaded performance. When the computational demands from both ALU and FPU units are high, the morphed core performs better than either one of the constituent ACMP cores. Hence, the morphed core is capable of handling sequential or parallel threads that have a variety in their instruction mix.

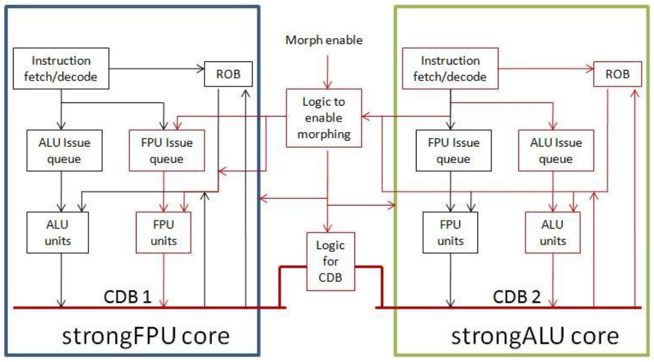

Fig. 1. Hardware support for morphing

IV. Hardware support for morphing

Core morphing involves a dynamic reconfiguration of the ACMP into a single core that is used for single threaded performance. In this section we provide an overview of the hardware support for morphing. In Fig. 2, the additional hardware support for core morphing is shown. In the usual mode of operation, morph-enable is deasserted and the two cores work independently of one another with no communication between them. When we detect that morphing is indeed necessary, either by detecting variations in the IPC or by determining a change in the composition of the workload being executed, morph-enable is asserted and the logic to enable morphing comes into play. It can be seen from Fig. 1 that the morphed core uses all the resources of the strongALU core and takes charge of the FPU units from the strongFPU core. The other unused units are power-gated. In the uni-processor mode, instructions are fetched in the strongALU core and FPU instructions are dispatched to the issue queue in the strongFPU core. Logic is needed here to inform the strongALU core that it is now in the uni-processor mode and has access to the FPU units of the strongFPU core. In modern OOO processors, the Reorder Buffer (ROB) is used to enable in order commit for precise interrupts. Since in the uni-processor mode the FPU units of the neighboring strongFPU core are used, there must be a provision to ensure that the results of the FPU units in the strongFPU core make it to the ROB of the strongALU core. At the same time, data needed by the FPU units must be fetched from the ROB in the strongALU core and not that in the strongFPU core.

This logic for the instruction issue and ROB can be easily implemented by means of a multiplexer or a tri-state driver. For the sake of brevity, we show the logic for just the passing of results from the Common Data Bus (CDB) of the strongFPU to that of the strongALU core. In the tri-state driver based design, the driver is controlled by the morph-enable signal such that data is passed from one CDB to the other. In the multiplexer based design, the inputs for the 2:1 multiplexer come from the outputs of the FPU units of the two cores and the select is the morph-enable. Considering 32-bit FPU outputs, the multiplexer is a 2 select 32-bit input multiplexer. In either case, whether tri-state or multiplexer based, we need two of each, one for the instruction issue and the other for the ROB. An additional overhead is the interconnect, and for a multimillion gate dual-core processor we estimate this overhead to be less than 1%.

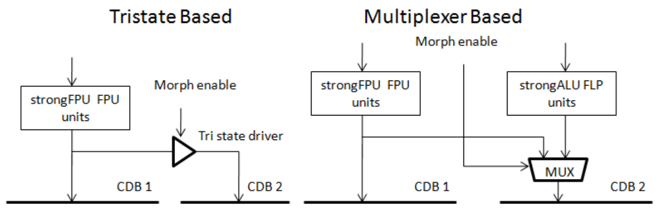

Fig. 2. Possible hardware solutions for data passing in the uni-processor mode

V. Experiments to determine processor parameters

The design space for processor cores is particularly large, and our goal was to find a set of processor parameters that best fits the role of the strongALU and strongFPU cores. In this section, the experiments used to set the parameters for each core are described.

For our experiments we used the SimpleScalar architectural simulator [16] on a class of benchmarks from SPEC2000 [7], Mediabench [14] and the Mibench [15] embedded benchmark suites. The benchmarks were chosen to target a wide range of applications. Some of these could be high-end scientific computations, embedded applications like secure hashing, and media applications like jpeg encoding/decoding.

As mentioned earlier, we consider for our experiments two cores: one with a strong floating-point unit (and weak integer unit) and the other with a strong integer unit (and a weak floating-point unit). We call these two cores StrongFPU and StrongALU, respectively. The experimental methodology consists of determining the best configurations for the two cores to suit the majority of the benchmarks considered. This will ensure that we are not undersizing the resource while studying the potential benefits of core morphing. The strategy for determining the configuration for a core was to select a configuration such that the majority (90%) of floating-point intensive and integer intensive benchmarks will run with reasonable performance on the StrongFPU and the StrongALU core, respectively.

A. Determining the base configuration

In the first set of experiments we determined the best possible configuration for the individual cores. We selected 24 different benchmarks with 15 from the SPEC2000 suite, 3 from Mediabench (MeB) and 6 from Mibench (MiB). The configurations are determined by varying the following parameters: (i) L1 Cache Size (D and I), (ii) L2 Cache Size (D and I), (iii) ROB

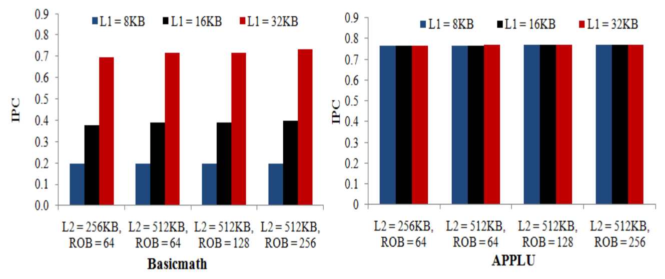

Fig. 3. IPC performance for APPLU and Basicmath

size and (iv) LSQ (Load/Store Queue) size.

Fig 3 shows the result of varying these parameters for APPLU and Basicmath running on the strongFPU and strongALU cores, respectively. The IPC improvement for APPLU going from the leftmost bar (L1 = 8 KB, L2 = 256 KB and ROB = 64) to the rightmost bar (L1 = 32KB, L2 = 512 KB, ROB = 256) is 1%, which is insignificant. Clearly, APPLU can be run on the StrongFPU core with L1 = 8KB, L2 = 256 KB and ROB = 64 without significant drop in performance. We did similar experiments for 8 floating-point intensive benchmarks (APPLU, WUPWISE, SWIM, MGRID, AMMP, APSI, ART, and EQUAKE). Apart from AMMP and EQUAKE, none of the other benchmarks showed noticeable improvement from a bigger L2 cache (512 KB) and a larger ROB (256 entries). The best configuration for the strongFPU core is shown in Table I. The table also lists the parameters for the backend section (FP and INT execution units) of the strongFPU core. The strongFPU core is designed with a pipelined (strong) floating-point execution unit and a non-pipelined (weak) integer execution unit.

A similar set of experiments were performed for the integer intensive benchmarks. Fig 3 shows the IPC variation for the Basicmath benchmark from the Mibench embedded benchmark suite. The IPC improvement in going from the leftmost bar (L1 = 8 KB, L2 = 256 KB and ROB = 64) to the 6th bar (L1 = 32 KB, L2 = 256 KB and ROB = 128) is 260%, which is significant. Moreover, going to the rightmost bar (L1 = 32KB, L2 = 512 KB and ROB = 256), there is an additional improvement of only 2%, not significant enough to justify a bigger L2 of size 512 KB and a big ROB of size 256. The suitable configuration of the strongALU core to run the Basicmath application is therefore L1 = 32KB, L2 = 256KB and ROB = 128. Out of the 16 integer intensive benchmarks, 13 benchmarks show a similar behavior. For the remaining three benchmarks (mcf, vortex and sha), a configuration of L1 = 8KB, L2 = 256 KB and ROB = 64) is sufficient. The system parameters for the strongALU core are also shown in Table I. The backend section of the core includes a pipelined (strong) INT execution unit and a non-pipelined (weak) FP execution unit. In the uni-processor mode of operation, the bigger frontend and the pipelined (strong) INT execution unit of the strongALU core will be used along with the pipelined (strong) FP execution unit of strongFPU core.

TABLE I
CORE PARAMETERS

StrongFPU core parameters						
Front end						
F/I/R 2	L1 D$ 8K	L1 I$ 8K	L2 $ 256K	L2 I$ 256K	ROB 64	LSQ 32
Back end						
FP units				INT units		
ALU	MUL	DIV		ALU	MUL	DIV
2 units 4 cycles	1 unit 4 cycles	1 unit 12 cycles		1 unit 2 cycles	1 unit 30 cycles	1 unit 120 cycles
StrongALU core parameters						
Front end						
F/I/R 2	L1 D$ 32K	L1 I$ 32K	L2 $ 256K	L2 I$ 256K	ROB 128	LSQ 64
Back end						
FP units				INT units		
ALU	MUL	DIV		ALU	MUL	DIV
1 units 10 cycles	1 unit 30 cycles	1 unit 120 cycles		2 unit 1 cycles	1 unit 3 cycles	1 unit 12 cycles

TABLE II
CLASSIFICATION OF THE VARIOUS BENCHMARKS

Benchmarks	Suite	Class	Benchmarks	Suite	Class
wupsize	SPEC	C	twolf	SPEC	C
equake	SPEC	C	vortex	SPEC	C
apsi	SPEC	C	cjpeg	MeB	C
bzip	SPEC	C	djpeg	MeB	C
gcc	SPEC	C	mpeg2enc	MeB	C
parser	SPEC	C	sha	Mib	C
gzip	SPEC	C	dij	Mib	C
mcf	SPEC	C	patricia	Mib	C
crc	MiB	C	basicmaths	Mib	C
applu	SPEC	A	mgrid	SPEC	A
ammp	SPEC	B	swim	SPEC	B
art	SPEC	B	fft	MiB	B

VI. RESULTS

In this section we show the benefits of our method with respect to IPC for various applications when running in the uni-processor mode. We also show the potential benefits of morphing by running two threads in parallel on the individual cores and then back to back on the morphed core, varying the load for each thread and comparing execution times. For the SPEC benchmarks we ran 10 million instructions after skipping the initialization instructions, while for the embedded benchmarks we ran 1 billion instructions without skipping any instructions.

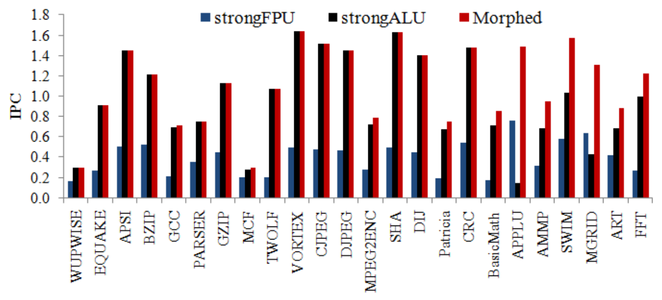

Fig. 4. IPC for the benchmarks considered on the three cores

A. IPC comparison results

After selecting the best configuration for the strongFPU and strongALU core, we performed a set of experiments on the 24 benchmarks to determine the potential benefits of core morphing. In these experiments, we assume that the FP execution units in the uni-processor mode of operation require an extra cycle latency to account for the overhead of morphing. The IPC results are shown in Fig 4. Out of the 24 benchmarks considered, only 6 show improvement from morphing of cores, namely, APPLU, MGRID, SWIM, AMMP, ART and FFT. Based on these results we classified the 24 benchmarks into three categories, as shown in Table II. Class A includes the benchmarks which run decently on the strongFPU core and show improvement from morphing. Class B consists of benchmarks which run well on the strongALU core and show improvement from morphing. Class C comprises of those benchmarks which do not show any improvement from morphing.

Out of the 6 benchmarks that belong to either class A or B, only 2, namely, APPLU and MGRID, show a considerable speedup of almost 100% (when moving from the segregated to the morphed core), while the remaining 4 benchmarks (SWIM, AMMP, ART and FFT) show a smaller speedup (10-50%). This can be explained by looking at the temporal distribution of instructions for the benchmarks. To illustrate this we show in Fig. 5 the distribution for APPLU. Fig 4 shows that APPLU benefits significantly (100%) when moving from the strongFPU to the morphed core. The improved performance can be attributed mostly to the presence of the strong INT units in the morphed core. Fig. 5 shows the frequency of a given percentage of INT instructions within a window of a certain size, for two window sizes, namely 200 and 1000. For both curves, the total number of APPLU instructions executed is 10 millions. The average percentage of INT instructions for APPLU during the execution of all its 10 million instructions is 48%, but it can be seen that when the window size is reduced, the variance increases considerably. This means that for several narrow windows of execution the percentage of INT is much higher than the global average and in such situations the performance provided by the weak INT unit is no longer adequate and a strong INT unit can be very beneficial. We have similarly analyzed the temporal distribution of the frequency of INT instructions for the ART benchmark that has a similar overall average frequency of INT instructions. The variance changes very little when the window size is reduced and as a result, ART does not benefit significantly from morphing. Similar behavior has been observed for MGRID and SWIM.

B. IPC of multithreaded workloads

In this set of experiments, we considered parallel threads of two benchmarks and measured the performance improvement in the uni-processor mode over the segregated mode. The results are presented for 100 million cycles of execution. In the segregated mode of operation, one of the benchmarks goes through all its 100 million cycles while the runtime for the

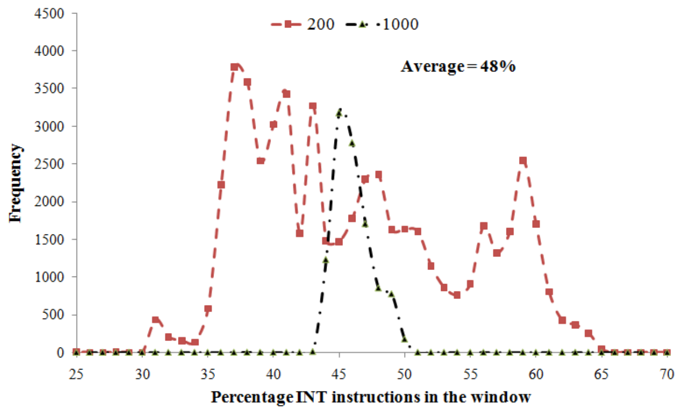

Fig. 5. Percentage distribution of INT instructions for APPLU

other benchmark varies from 0 to 100 million cycles (in steps of 10 million cycles). Thus, the overall runtime is 100 million cycles. In the uni-processor mode, the benchmarks execute on the morphed core in an interleaved manner. The runtime of one of the benchmarks varies from 0 to 100 million cycles (in steps of 10 million cycles) while the runtime of the other benchmark varies from 100 million cycles to 0 (in steps of 10 million cycles). The total execution time remains 100 million cycles.

Fig 6 shows the results for one such multithreaded workload consisting of the benchmarks FFT (integer intensive) and MGRID (floating-point intensive). In the segregated mode, MGRID is executed for 100 million cycles and the load of FFT is varied. In the uni-processor mode, FFT and MGRID are executed in an interleaved manner. The execution time of FFT increases from 0 to 100 million cycles while the execution time of MGRID decreases from 100 million cycles to 0 with the same step so as to keep the total execution time constant and equal to 100 million cycles. The results for the two modes are shown as two separate curves in Fig 6. The curves cross each other at around 60%. This implies that if the execution time of FFT is below 60% of the total execution time, morphing will result in a higher total number of instructions executed. If, on the other hand, the execution time of FFT is more than 60% of the total execution time, a segregated mode will have more instructions executed.

Similar experiments were performed for all combinations of benchmarks from the three different classes A, B & C. For benchmark combinations where both prefer to run on the same core, a judicious decision was taken. For example, for the combination of MGRID and APPLU (both of Class A), where both are floating-point intensive benchmarks preferring the strongFPU core, in the segregated mode of operation, APPLU was mapped onto the strongFPU core and MGRID onto the strongALU core. This was done because mapping APPLU on the strongALU will degrade the IPC by 82% whereas mapping MGRID on the strongALU will degrade the IPC by only 31%. The results of some of the combinations of benchmarks are presented in Fig 7. For a combination like

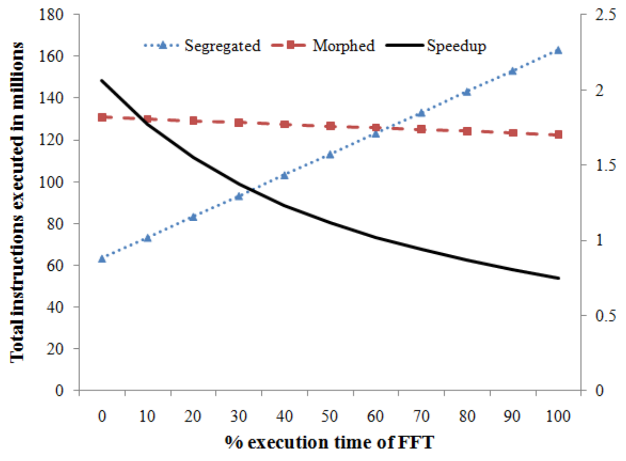

Fig. 6. Multithreaded workload of MGRID and varying load of FFT

MGRID + % APPLU, it is always beneficial to interleave the two threads and run them on the morphed core as the speedup is greater than one even for 100% execution time of APPLU. For combinations like SHA + % AMMP, the speedup is less than one for any variation of the execution time of AMMP. Thus, for a multithreaded workload consisting of SHA and AMMP, core morphing will not be beneficial. For other combinations, the curve crosses a speedup of one at some % of the execution time of the second benchmark. Therefore, for workloads that deliver higher IPC on a morphed core, thread interleaving on the morphed core outperforms the parallel execution of the two threads on the individual cores.

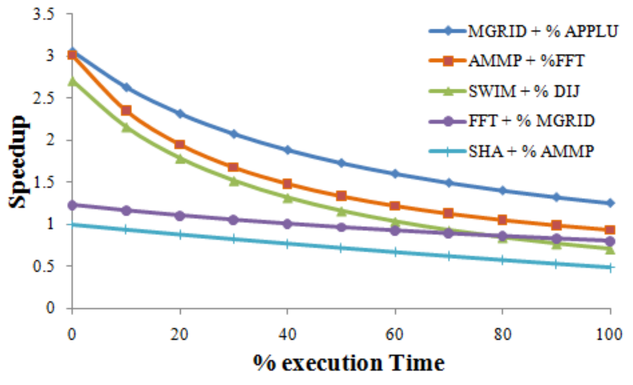

Fig. 7. Speedup for various combinations of benchmarks running on the morphed and segregated cores

VII. Conclusions

We conducted a study on the possible performance benefits of core morphing in an ACMP. A dual-core system was used for experimentation in which the cores were chosen such that one of them supports FP operations while the other ALU operations. The design of each core and the necessary hardware support have been discussed. Our results indicate that significant performance benefits (up to 100%) are possible in some cases while operating in the uni-processor mode. The benefits of the proposed architecture are evident from the experiments where the workload was varied showing that the morphed core (running the two threads back to back) outperforms the parallel execution of the two threads on the separate cores. We conclude that a reconfigurable ACMP can achieve higher performance for sequential/parallel workloads of various flavors.

Acknowledgment

This work has been supported in part by a grant from SRC (Grant no. 1985.001) and NSF (Grant no. 0903191).

References

[1] S. Ghiasi and D. Grunwald, “Aide de camp: Asymmetric dual core design for power and energy reduction,” University of Colorado Technical Report CU-CS-964-03, 2003
[2] P. Salverda and C. Zilles, “Fundamental performance constraints in horizontal fusion of in-order cores,” In Proceedings of the Int’l Symposium on High Performance Computer Architecture (HPCA), 2008.
[3] R. Kumar et al., “Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction,” In Proceedings of the 36th Annual IEEE/ACM international Symposium on Microarchitecture, December 2003.
[4] R. Kumar et al., “Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance,” In Proceedings of the 31st Annual International Symposium on Computer Architecture, June 2004.
[5] S. Balakrishnan et al., “The Impact of Performance Asymmetry in Emerging Multicore Architectures,” SIGARCH Comput. Archit. News 33, 2, May 2005, pp. 506-517.
[6] E. Grochowski et al., “Best of Both Latency and Throughput,” In Proceedings of the IEEE international Conference on Computer Design (ICCD), October 2004, pp. 236-243.
[7] The standard Performance Evaluation Corporation. Spec CPI2000 suite. http://www/specbench.org/osg/cpu2000.
[8] E. Ipek et al., “Core fusion: accommodating software diversity in chip multiprocessors,” In Proceedings of the 34th Annual international Symposium on Computer Architecture (ISCA ’07), June 2007, pp. 186-197.
[9] H. H. Najaf-abadi et al., “Core-Selectability in Chip Multiprocessors,” 18th International Conference on Parallel Architectures and Compilation Techniques, 2009, pp. 113-122.
[10] C. Kim, Sethumadhavan et al., “Composable Lightweight Processors,” In Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture, December 2007.
[11] R. Kumar et al, “Conjoined-Core Chip Multiprocessing,” In Proceedings of the 37th IEEE/ACM international Symposium on Microarchitecture, December 2004, pp. 195-206.
[12] T. Morad et al., “ACCMP - asymmetric cluster chip-multiprocessing,” In CCIT Technical Report 488, 2004.
[13] R. Kumar et al., “Core architecture optimization for heterogeneous chip multiprocessors.,” In Proceedings of the 15th international Conference on Parallel Architectures and Compilation Techniques, September 2006, PACT ’06.
[14] C. Lee et al., “MediaBench: a tool for evaluating and synthesizing multimedia and communications systems,” In Proceedings of the 30th Annual ACM/IEEE international Symposium on Microarchitecture, December 1997.
[15] M. R. Guthaus et al., “MiBench: A free, commercially representative embedded benchmark suite,” In Proceedings of the IEEE Workload Characterization, December 2001.
[16] D. Burger, and T. Austin, “The SimpleScalar tool set,” version 2.0. SIGARCH Comput. Archit. News 25, 3, June 1997, pp. 13-25.
[17] D. Wall, “Limits of instruction-level parallelism,” SIGARCH Comput. Archit. News 19, 2, Apr. 1991, pp. 176-188.