Exploring ILP and TLP on a Polymorphic VLIW Processor

Anthony Brandon, Joost Hoozemans, Jeroen van Straten, and Stephan Wong

Computer Engineering Lab, Delft University of Technology, The Netherlands
{a.a.c.brandon, j.j.hoozemans, j.vanstraten-1, j.s.s.m.wong}@tudelft.nl

Abstract. In today's computing environments, the concurrent execution of multiple applications/threads is common and multi-cores are very well-suited to handle such workloads. However, they suffer from the fact that any mismatch between the application's inherent instruction-level parallelism (ILP) and the core's parallelism leads to unused resources or loss in performance. An accepted solution is to include several types of cores and match them dynamically depending on the performance needs of the application. This approach becomes less efficient when the number of cores does not match the number of parallel threads. Furthermore, the heterogeneity of (fixed) cores cannot be increased indefinitely as it would result in even higher degrees of mismatching and increased movement of instruction and data streams. In this paper, we are proposing a polymorphic processor, based on VLIW architectures, that can adapt its issue-width during runtime. By design, the processor can be perceived as a single wide core (8-issue VLIW) or two medium-wide cores (4-issue) or four small cores (2-issue) that can run high-ILP/low DLP, medium-ILP/medium DLP, and low-ILP/high-DLP applications, respectively. Furthermore, we are executing one single generic binary while performing these reconfigurations. In order to show the effectiveness of our approach, we synthesized different versions of the core to represent fixed heterogeneous cores and compared them to the dynamic implementation of the core. Our experiments show that the dynamically adaptive solution performs on average 7% faster and uses 5% less area than a platform which consists of fixed cores with 1.5× as many datapaths.

1 Introduction

Modern embedded systems (including smartphone SoCs and other low-power, high-performance systems) are faced with dynamic workloads [1]. Workloads can be dynamic in several characteristics, e.g., performance requirements of tasks, instruction-level parallelism (ILP), data-level parallelism (DLP), or task-level parallelism (TLP). The state-of-the-art in technology in the embedded low-power high-performance domain addresses these workloads using heterogeneous multi-core processing systems, e.g., Exynos that adopts the big.LITTLE approach [2]. These allow different types of tasks to be mapped to a processing unit that most closely matches their characteristics and/or requirements. As the processing elements of a heterogeneous system are fixed, the performance of the system

depends on the extent to which the workload can be mapped to the hardware (called *performance fragility* by [3]). Previous research [3], [4], [5], [6], [7], and [8] presented polymorphic architectures as a way to address this problem. A *dynamic* hardware platform, that is able to change its characteristics at runtime, can provide high performance for single-threaded applications by exploiting ILP and high throughput for multi-threaded applications by exploiting TLP [9].

Until now, no-one has quantified the potential benefits of a dynamic hardware platform over fixed platforms. We pose the following question: given the same number of transistors, what is the best possible use of them to provide maximum performance under workloads with varying amounts of ILP and TLP exhibited by modern multi-threaded programs?

In this paper, we introduce and evaluate our proof-of-concept for a dynamic approach to run these dynamic workloads in the high-performance embedded domain. We apply polymorphism, that has been applied previously to high-performance general-purpose designs, to a Very Long Instruction Word (VLIW) processor that is more suitable for low-power systems. The result is a dynamic system that can adapt its processor and cache to the running applications. Note that our "dynamic" approach does not rely on (partial) FPGA reconfiguration. Our designs are written in synthesizable VHDL and prototyped on FPGAs, but are also suitable for implementation on ASIC.

We show that our dynamic approach is able to achieve a more efficient utilization of the available hardware resources. We are able to achieve on average 7% better performance using a dynamically reconfigurable processor when compared to a static system that is roughly 10% larger in area and has 1.5× as many datapaths.

2 Related Work

There are several existing attempts at making reconfigurable processors in order to exploit both TLP and ILP. Some, such as Core Fusion [5], Trips [3], and MorphCore [4] combine multiple small cores into large superscalar (possibly Out-of-Order) cores to exploit the parallelism available at runtime. While these approaches have binary compatibility between the different configurations, they require expensive and power-hungry hardware to extract ILP from the (sequential) code at runtime. Moreover, most of these approaches were only simulated, whereas we have a working prototype on an FPGA.

Other attempts such as Voltron [6] and Smart Memories [7] are based on VLIW architectures, where smaller cores are combined to form a wide issue VLIW. MT-ADRES [10] is a Course Grain Reconfigurable Array (CGRA) which is similar to a VLIW, which can also be configured as either a single wide issue core, or multiple smaller cores. This is similar to our approach, however these do not have binary compatibility between the different configurations. This means that it is determined at compile time which parts of the program will operate in which configuration, giving less flexibility at runtime. Our approach does have this flexibility, which allows the core to be interrupted at any point in time,

reconfigure to a different issue-width (in only several clock cycles), and resume execution. Our design also achieves this without any need for state saving and/or context switching.

VLIW processors are an alternative to superscalar processors[11] with the goal of reducing power consumption by moving dependency checking to the compiler, thus simplifying the hardware (and in turn consuming much less power). One downside of fixed issue-width VLIW processors is that resources will go unused if the issue-width is greater than the ILP available in the program. On the other hand, if the issue-width is too low, potential performance gains are lost. Making the issue-width runtime reconfigurable addresses this problem.

The processor we target in this paper is a reconfigurable issue-width VLIW processor [12] based on ρ-VEX [13], a parameterized VLIW processor. The processor can be configured at design time to have an issue-width of 2, 4, or 8 and can be reconfigured at runtime to split into smaller processors with issue-widths of 2 or 4. We explore the platform's performance under different workloads with varying amounts of TLP in order to quantify the benefits of reconfigurability.

3 Approach

Our approach to increase the utilization of execution datapaths in a processor allows for **on-the-fly** composition of cores — the reconfiguration time is expressed in clock cycles as demonstrated in our FPGA prototype (see Section 4). Cores can be merged to "construct" wider cores for applications needing performance (with enough available parallelism). When this is not required, multiple applications can be executed in parallel. In addition, (low-priority) applications can be "forced" to execute on a smaller core allowing other applications to execute in parallel. Such a scenario is impossible for fixed processors when a second free processor is not available, or would require costly context-switching, effectively stalling the first application. Our approach is also different from reorganizing datapaths from RISC cores into multiple cores as such approaches would still require (several) power-hungry instruction decoders. Starting from a VLIW architecture, the need for such decoders can be omitted and the reconfiguration can be achieved by simply reassigning the datapaths and register file ports.

The versatility of our core design is achieved by separating the program contexts and the execution datapaths (pipelanes) and controlling their connections via a reconfiguration controller (as depicted in Figure 1):

- Pipelanes & lane-groups: functional units, instruction fetch and decode, etc.
- Contexts: the register file, and all state related registers, such as program counter, control registers, etc.

The VLIW core comprises multiple execution pipelines, also called pipelanes. For instance, an 8-issue ρ-VEX has 8 pipelanes. Subsequently, we divide these pipelanes into lane-groups. For instance pipelanes 0 and 1 form lane-group 0, and pipelanes 2 and 3 form lane-group 1, and so on. When two lane-groups are

joined, the processor functions as a 4-issue VLIW. When they are separate, they function as two independent 2-issue VLIWs.

The contexts contain the state of programs and since we can execute 4 programs in parallel at least 4 contexts are needed. Each lane-group can access each context and which context it accesses depends on the current configuration of the core. In this way, we can reconfigure the core by assigning contexts to lane-groups. The contexts are connected to lane-groups through a switch network which allows each lane to access each context. The register file is described in more detail in [14]. When combining lane-groups, wider-issue VLIW cores can be composed and they benefit from the same flexibility in choosing which context to execute. The added benefit of this approach is that multiple contexts are maintained and context switching among them can be achieved in just a few cycles — basically, flushing the associated pipelanes. In addition, "moving" a context to be executed on a differently-sized core (similar to moving a program from a low-performance core to a high-performance core or vice versa) requires only a reconfiguration of our core. Moreover, the I-caches in our design follow the reconfiguration of the core; therefore, their content is also maintained, thereby removing the need for code movement or a cold start of the I-cache when physically moving a program to another core.

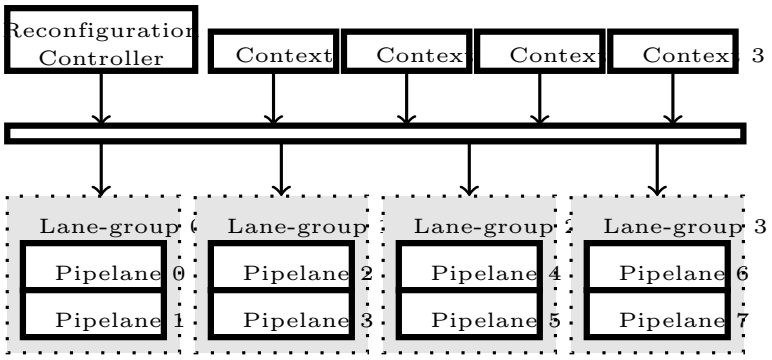

Fig. 1. Interconnect between register contexts and processor pipelines controlled by Reconfiguration Controller.

Because we are dynamically reconfiguring the core, we have to deal in more detail with control flow and interrupts. Each lane-group has its own branch unit and after merging all but one of them is disabled in order to simplify the instruction fetch hardware. When handling interrupts (including exceptions and traps), they must be handled differently depending on the configuration as they are tied to contexts. We designed the core to correctly stall the right context, invalidate any remaining load/store operations in case of a bus-fault, and flush the associated pipelanes before jumping to the trap handler.

Another important aspect of our approach is to maintain binary compatibility of the VLIW instructions when executing them on different issue-width cores. For this purpose, we use generic binaries [15] (reporting a performance hit between 10% and 30%) and employed techniques described in [16] to reduce the performance hit to on average around 5%.

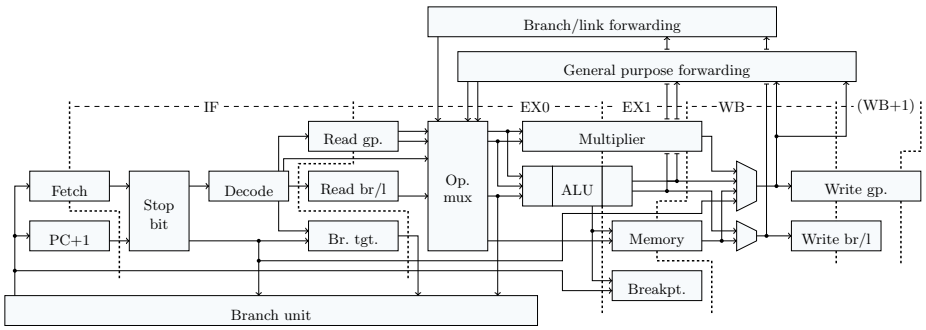

Fig. 2. Schematic of a single datapath of the processor.

4 Implementation

In Section 3, we described the requirements of the reconfigurable processor. In this section we explain how we implemented the reconfigurability in hardware. The design is implemented in VHDL and is completely parameterized using generics, so that it is possible to statically change the number and size of lane-groups that will be available at runtime. For example, instead of having an 8-issue ρ-VEX that can split into four 2-issue, or two 4-issue cores, we could have a 4-issue core that can split into two 2-issue cores. The number of contexts is also parameterized, making it possible to have more contexts than the maximum number of cores.

In Figure 2, we depict the pipeline stages of a single datapath of the processor. The processor consists of a (design time) configurable number of these datapaths, which are connected through the forwarding logic and the register files as depicted in Figure 1. Each datapath can be configured to have an ALU, branch unit, load/store unit, and multiplier.

As described in the previous section, the ρ-VEX can be reconfigured at runtime by assigning lane-groups to contexts. This is done by writing to the configuration control register. The control registers are part of a memory mapped address space, so this can be accomplished through a standard store instruction to the right address. Once a new value has been written to the context control register, the reconfiguration controller will check that the requested configuration is valid. For instance, in an 8-issue reconfigurable ρ-VEX, it is not possible

for the center two lane-groups to be merged into a single core. Similarly, it is impossible to assign multiple non-adjacent lane-groups to the same context.

Once the requested configuration has been determined to be valid by the reconfiguration controller, the ρ-VEX will stop fetching new instructions and will finish executing any in-flight instructions. Once the pipelanes are empty, the actual configuration takes place and each context continues execution where it was last stopped. Note that because all instructions in the pipeline must finish execution before reconfiguration completes, reconfiguration can take a variable number of cycles depending on the exact instructions being executed. The minimum number of cycles for a reconfiguration to take place is five cycles, while the maximum is the same as for a load instruction that causes a cache miss.

5 Experimental Results

In order to demonstrate the benefits of our approach we wanted to show its performance when executing multiple tasks with different performance characteristics on different configurations of the core. By reconfiguring the core at runtime we can adapt to the current workload and in doing so utilize our hardware resources more efficiently.

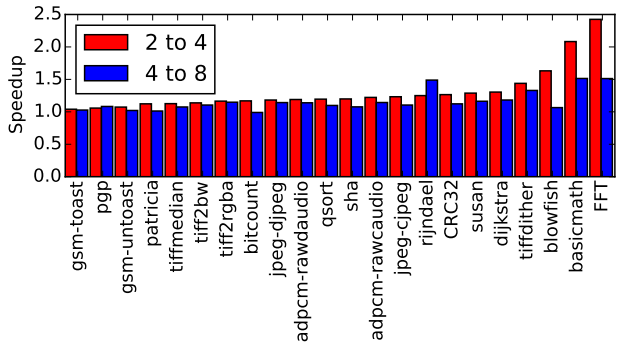

Fig. 3. The speedup for each benchmark from two to four issue, and from four to eight issue.

5.1 Workload definition

For our experiments we use workloads consisting of Mibench applications [17]. First, we characterized the individual applications in terms of available ILP and execution time. Figure 3 shows the speedup between executions on 2-issue and 4-issue, and between executions on 4-issue and 8-issue. From this figure we can observe that most applications have relatively little ILP, while a handful

have high ILP. The reason that FFT can achieve a greater speedup than 2, is because the cache of the 4-issue processor is larger. We also notice that none of the benchmarks have a very large speedup when switching to 8-issue. For this reason, we decided to focus on 4-issue and 2-issue configurations of the ρ-VEX in the remainder of this paper. We do this because implementing a large 8-issue VLIW incurs a large overhead in terms of area, which cannot be justified for these applications.

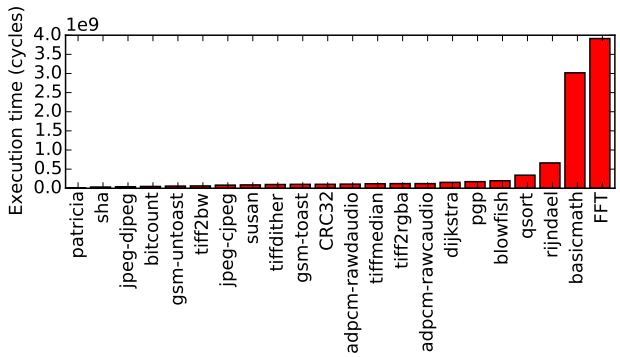

Fig. 4. Execution time for each benchmark.

In Figure 4, we depict the execution time of each benchmark in cycles. From this figure we can observe that two of the benchmarks are especially long compared to the others. Since we are interested in the performance of multiple threads running on the cores simultaneously, this is undesirable. If we select one of these extremely long benchmarks to run together with shorter benchmarks, the results will be dominated by the performance of a single application running most of the time. In order to mitigate this effect we normalized the execution times for each benchmark to the execution time. This means that we are only comparing the relative improvement from 2-, to 4-, to 8-issue.

In modern embedded devices it is usually not possible to know at design time the exact characteristics of the workload it will be running. For example, a smartphone might be running a single or multiple applications depending on what the user is doing. In order to understand how platforms consisting of various numbers of processor cores will perform under these different conditions we must construct workloads of varying numbers of applications to run simultaneously. Therefore, we randomly choose 1 to 22 different applications from the list of all our Mibench applications and run these simultaneously on all of the platforms. We then measure the time required to finish the entire workload. For each workload size (1 – 22) we created 100 random workloads (2200 total).

In addition to using these real world benchmarks, we also estimated the performance of each platform for a set of idealized synthetic benchmarks which

allows us to analyze the performance of the platforms under simplified conditions. These benchmarks exhibit ideal ILP which makes it easier to reason about how the system will perform.

5.2 Resource Utilization

We used Synopsis Design Vision to obtain area results for ρ-VEX on a 65 nm library, and CACTI [18] to obtain area estimates for caches for the different cores. Table 1 shows the parameters used to estimate area for the instruction caches. In this table, we have assumed that for a 4-issue core we will use a cache twice as large as for a 2-issue core. In order to represent the reconfigurability of the cache for the reconfigurable core we estimate the area as twice the size of a cache with a wider line size. Table 2 shows the area of the 2-, 4- and 4-issue reconfigurable cores, along with instruction and data caches. This table shows that the caches make up a significant amount of the area. Using these numbers we determine several multi-core platforms that fit in similar area.

Table 1. Instruction cache parameters

	4r	2	4
Cache Size (KiB)	8	8	16
Line Size (B)	16	8	16
Nr. of Banks	1	1	1
Associativity	1	1	1
Size (mm^2)	0.215×2	0.146	0.242

Table 2. Area results on 65 nm (mm^2).

	4r	2	4
Core	0.263	0.114	0.175
I-Cache	0.432	0.146	0.242
D-Cache	0.292	0.146	0.270
Total	0.987	0.407	0.688

5.3 Experimental Setup

In order to evaluate our approach, we compared our platform consisting of 2 reconfigurable 4-issue processors to other platforms consisting of multiple static processors of equivalent total area. This resulted in the following platforms:

- 44r (1.974mm^2): The dynamic platform consists of two 4-issue reconfigurable cores. This means that this platform can execute two tasks in parallel on the 4-issue cores, or three tasks in parallel with one on a 4-issue, and two on 2-issue cores, or it can execute four tasks in parallel on 2-issue cores.
- 444 (2.065mm^2): This configuration consisting of three static 4-issue cores is the largest in terms of computational resources that fits in roughly the same area as two 4-issue reconfigurable cores.
- 2222 (1.623mm^2): This platform can achieve the same maximum TLP as the dynamic platform, namely four threads at once, which makes for a more interesting comparison.

– 442 (1.784mm^2): We use this configuration to represent a heterogeneous system similar to big.LITTLE with a mix of cores of different issue-widths. This would allow for efficient mapping of tasks to cores when dealing with a mix of tasks with high and low ILP.

In each of these platforms, the 2-issue cores have an instruction and data cache size of 32 KiB while the 4-issue cores have cache sizes of 64 KiB. The reconfigurable cores have 64 KiB caches each when operating in 4-issue mode, or 32 KiB caches for each core when operating in 2-issue mode.

In order to obtain execution times these platforms were all synthesized for the Xilinx Virtex 6 FPGA running at 37.5 MHz. We use GRLIB [19] for peripherals such as UART output, interfaces to DDR memory, and the interrupt controller.

Listing 1.1. Scheduler for static platforms.

```
# inputs:
# readylist: list of all tasks sorted by
#            descending ILP
# cores: list of processor cores sorted by
#        descending issue-width
while len(readylist) > 0:
    while len(running) < len(cores):
        running.append(readylist.pop())
    running.sort()
    for i in len(running):
        cores[i].task = running[i]
    wait()
```

Listing 1.2. Scheduler for reconfigurable platforms.

```
# inputs:
# readylist: list of all tasks sorted by
#            descending ILP
while len(readylist) > 0:
    while len(running) < 4:
        running.append(readylist.pop())
    running.sort()
    if len(running) == 4:
        cores.config = [2,2,2,2]
    elif len(running) == 3:
        cores.config = [4,2,2]
    elif len(running) == 2:
        cores.config = [4,4]
    else:
        cores.config = [4]
    for i in len(running):
        cores[i].task = running[i]
    wait()
```

In order to run the various workloads as defined in Section 5.1 we implemented a simple task scheduler, which runs on the processor. It decides which task to run on which core, and in the case of the dynamic core, in which configuration. The scheduler for the fixed platforms, shown in Listing 1.1, uses a greedy approach to run as many tasks in parallel as possible. The input for this scheduler is the list of tasks which have to be executed, and the issue-width of each available core. The list of tasks is ordered by descending ILP to ensure that tasks with high ILP will be scheduled on the cores with the highest issue-width. If a task is running on a 2-issue core and a 4-issue core becomes available, the

scheduler will migrate that task to a larger core. After the tasks are assigned to cores the scheduler waits for one of the tasks to complete, at which point it will run again to schedule any remaining tasks. This continues until all the tasks have finished.

The scheduler for the dynamic platform, shown in Listing 1.2, is slightly different because it not only has to distribute the tasks to the cores, but also determine the desired configuration. The scheduler first selects the maximum number of tasks to run, under the assumption that it is, in most cases, more efficient to run multiple tasks than it is to run a single task on a high issue-with. Next, depending on the number of tasks, the scheduler chooses one of the possible configurations. For example, if there are three tasks to run, it will choose the *422* configuration. It then schedules the task with the highest ILP on the largest core than can be formed.

These two schedulers assume that the average ILP of each task is known beforehand in order to make comparisons in performance assuming the best possible schedules. Techniques for scheduling based on measuring ILP at runtime are out of scope, and are discussed in other work [20].

5.4 Performance of Synthetic Workloads

We will now examine the performance of each of the chosen platforms using artificial workloads tailored to each specific platform. The workloads consist of as many tasks as there are cores in the system, with the ideal ILP for that particular mix of cores. Figure 5 shows how these synthetic workloads could be scheduled on each platforms. Each color represents a task and the height of a block represents execution time, while the width represents the issue-width it is executed on. For a platform with four 2-issue cores we use four tasks, each with an ILP of 2. On the platforms with fewer than four cores the fourth task has to be executed after the other tasks, resulting in longer total execution time. For the *444* platform the workload consists of three tasks with an ILP of 4. As shown in Figure 5, when this workload is executed on the *44r* platform, the three tasks are executed in parallel, and when one is done, the remaining two are switched to run in 4-issue mode. Our goal in using these synthetic benchmarks was to show that while the reconfigurable platform might not be the best in every single case, it is not bad either, and for the average case performs well.

The synthetic results in Table 3 show that the average execution time of the *444* platform is the same as for the *44r* platform. The *444* platform performs well for all cases where there are three or less threads, however when there are four or more threads the *44r* platform performs better. The *44r* platform performs worse when there are three high ILP tasks, however, since it can run three tasks in parallel it will only run 50% longer in the worst case. Real world applications do not exhibit this kind of difference between 2- and 4-issue, as depicted in Figure 3, which means that usually the reconfigurable platform would take less than 50% longer for a workload of three high ILP tasks.

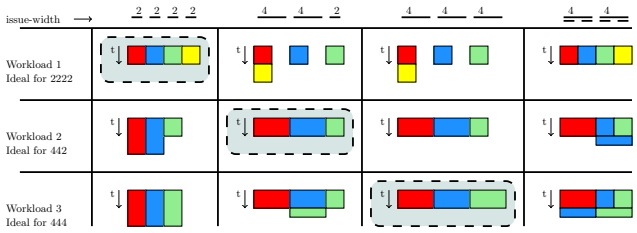

Fig. 5. Three synthetic workloads scheduled onto the four platforms being compared. Each color represents a task, while height represents execution time and width represents the issue-width of the processor executing that task.

Table 3. Synthetic workload execution times.

	44r	442	444	2222
Workload 1	1	2	2	1
Workload 2	1.5	1	1	2
Workload 3	1.5	1.5	1	2

Table 4. Average execution times and standard deviation of all Mibench workloads.

	44r	442	444	2222
relative execution time (mean)	1.0	1.11	1.07	1.09
standard deviation	0.03	0.06	0.06	0.0

5.5 Performance of Mibench Workloads

We also performed experiments with actual applications as explained in Section 5.1. The results, depicted in Figure 6, show that the reconfigurable platform is always at least as good as the *2222* platform, as expected. Remember that the *2222* platform is larger than the reconfigurable platform. When comparing with the *444* platform we can observe that there are some situations where the reconfigurable platform is better, and some cases where it is worse. However, note that there are fewer cases where the reconfigurable platform is slower, and that the difference in performance is not as high as in the cases where the *44r* platform is faster. The *444* platform is also roughly 5% larger than the reconfigurable platform. The average execution times and standard deviation in execution time are summarized in Table 4. We can observe that the reconfigurable platform is on average 7% faster than the static *444* platform. Additionally, we notice that the standard deviation is smaller for the dynamic platform. This means that the difference between workloads that perform well and workloads that perform poorly are larger on the static platform. We also observe that the *2222* platform has barely any deviation, however this means it has consistently poor performance. These results show that although the *444* platform has more total computational resources, and more available total cache, it cannot always make optimal use of the available hardware, because it cannot make the trade off between thread level parallelism and instruction level parallelism.

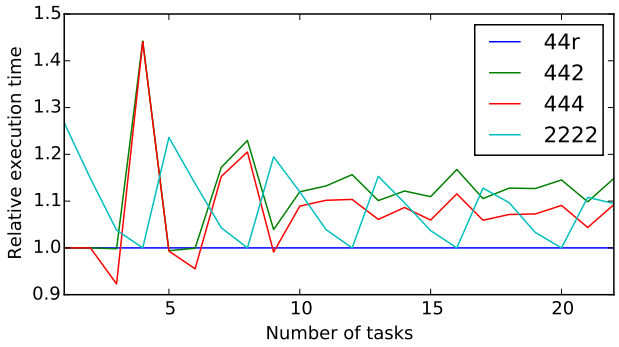

Fig. 6. Results of executing Mibench on different platforms with execution times normalized to 2-issue.

6 Conclusion

In this paper, we compared the performance of heterogeneous multi-core platforms to that of a dynamically reconfigurable platform. We did so using ρ-VEX, a reconfigurable VLIW processor with a reconfigurable cache, which is capable of reconfiguring at runtime to exploit either task-level parallelism, or instruction-level parallelism. We demonstrated that, while for a single workload a particular static configuration might be optimal, the reconfigurable platform can adapt to the workload and on average can more efficiently exploit the available hardware. Our results show that the reconfigurable platform is on average 7% faster than a static platform with 1.5× as many datapaths, and 5% larger area. Furthermore, in terms of performance the reconfigurable platform has more stable performance (less deviation) than other platforms for a wide variety of workloads.

References

1. C. H. van Berkel, "Multi-core for mobile phones," in *2009 Design, Automation Test in Europe Conference Exhibition*, pp. 1260–1265, April 2009.
2. P. Greenhalgh, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," *ARM White paper*, pp. 1–8, 2011.
3. K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, and C. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous trips architecture," *Micro, IEEE*, vol. 23, pp. 46–51, Nov 2003.
4. K. Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt, "Morphcore: An Energy-efficient Microarchitecture for High Performance ILP and High Throughput TLP," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pp. 305–316, IEEE, 2012.
5. E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 186–197, ACM, 2007.

6. H. Zhong, S. Lieberman, and S. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 25–36, Feb 2007.
7. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 161–171, June 2000.
8. R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu, "Improving Performance Per Watt of Asymmetric Multi-core Processors via Online Program Phase Classification and Adaptive Core Morphing," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, pp. 5:1–5:23, Jan. 2013.
9. S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The Impact of Performance Asymmetry in Emerging Multicore Architectures," *SIGARCH Comput. Archit. News*, vol. 33, pp. 506–517, May 2005.
10. K. Wu, A. Kanstein, J. Madsen, and M. Berekovic, *MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture*, pp. 26–38. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.
11. B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, overview, and perspective," *The Journal of Supercomputing*, vol. 7, no. 1, pp. 9–50, 1993.
12. F. Anjam, M. Nadeem, and S. Wong, "Targeting Code Diversity with Run-time Adjustable Issue-slots in a Chip Multiprocessor," in *Proc. Design, Automation and Test in Europe*, (Grenoble, France), March 2011.
13. S. Wong and F. Anjam, "The Delft Reconfigurable VLIW Processor," in *Proc. 17th International Conference on Advanced Computing and Communications*, (Bangalore, India), pp. 244–251, December 2009.
14. J. Hoozemans, J. Johansen, J. V. Straten, A. Brandon, and S. Wong, "Multiple Contexts in a Multi-ported VLIW Register File Implementation," in *Proc. 2015 International Conference on ReConFigurable Computing and FPGAs*, (Mayan Riviera, Mexico), December 2015.
15. A. Brandon and S. Wong, "Support for Dynamic Issue Width in VLIW Processors using Generic Binaries," in *Proc. Design, Automation & Test in Europe Conference & Exhibition*, (Grenoble, France), pp. 827 – 832, March 2013.
16. A. Brandon, J. Hoozemans, J. V. Straten, A. F. Lorenzon, A. L. Sartor, A. Beck, and S. Wong, "A Sparse VLIW Instruction Encoding Scheme Compatible with Generic Binaries," in *Proc. 2015 International Conference on ReConFigurable Computing and FPGAs*, (Mayan Riviera, Mexico), December 2015.
17. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, IEEE, 2001.
18. S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," *HP Laboratories*, 2008.
19. "LEON/GRLIB." http://www.gaisler.com/index.php/downloads/leongrlib. [Online; accessed 7-Sept-2016].
20. Q. Guo, A. L. Sartor, A. Brandon, A. Beck, X. Zhou, and S. Wong, "Run-time Phase Prediction for a Reconfigurable VLIW Processor," in *Proc. Design, Automation and Test in Europe*, (Dresden, Germany), pp. 1634 – 1639, March 2016.