# Support for Dynamic Issue Width in VLIW Processors using Generic Binaries

Anthony Brandon, Stephan Wong
Computer Engineering Laboratory, Faculty of EEMCS
Delft University of Technology, The Netherlands
Email: {A.A.C.Brandon, J.S.S.M.Wong}@tudelft.nl

***Abstract*—Different applications exhibit different behavior that cannot be optimally captured by a fixed organization of a VLIW processor. However, through exploitation of reconfigurable hardware we can optimize the organization when running different applications. In this paper, we propose a novel way to execute the same binary on different issue-width processors without much hardware modifications. We propose to change the compiler and assembler to ensure correct results. Our experiments show an *average* slowdown of around $1.3\times$ when compared to binaries compiled for specific issue-widths. This can be further improved to less than $1.09\times$ on average with additional compiler optimizations. Even though the flexibility comes at a price, it can be exploited for many other purposes, such as: dynamic performance/energy trade-off and energy-saving mechanisms, dynamic hardware sharing, and dynamic code insertion for hardware fault detection mechanisms.**

## I. Introduction

In the embedded domain, power consumption is a major design concern and in turn led to the use of Very Long Instruction Word (VLIW) processors for specific applications due to their inherent low-power requirements (no need for complex instruction scheduling in hardware) and capability to support applications with a high Instruction Level Parallelism (ILP) [1], [2]. Even though these are major advantages in the embedded domain, they are also the limiting factors that prohibited the wide-spread use of VLIW processors in the general-purpose domain, where the Itanium [3] is the only known example. In the latter domain, different applications have different characteristics and each one would require a different processor organization, which is impossible when considering that the designs are usually fixed. This fixed nature leads to inefficient resources utilization, higher power consumption, and reduced performance. On the other hand, Reduced Instruction Set Computer (RISC) processors can better handle the diversity of required resources of different applications, but this is at the expense of a complex instruction decoder that consumes a lot of power at every clock cycle. In this paper, we introduce a new approach to exploit reconfigurable hardware to bridge this gap by using VLIW processors to remove the need for a complex instruction decoder, but introduce a simple manner to quickly and, more importantly, dynamically match hardware resource utilization (and therefore also power) with available ILP and basically dynamically "ride" the performance/power trade-off curve per application. More specifically, we avoid the generation of multiple binaries for the different processor organizations, which would introduce the following disadvantages: (1) loading of instructions after switching to a different issue-width (consuming a lot of power in the instruction cache) and (2) switching between different versions of the code after interrupts (some method is required to ensure equivalent processor state before and after switching). Instead, we generate a single *generic binary*[1] (GB) that can be executed on cores with different issue-widths, which can be changed at run-time. Furthermore, this approach is orthogonal to existing power saving techniques such as clock gating [4] and voltage and frequency scaling [5].

Our main contribution in this paper is a way to split VLIW instruction bundles into smaller bundles while maintaining correct execution, with little additional hardware. This approach allows for the dynamic change of the issue-width during application execution, without the need for code reloading or complex mechanisms to support interrupts. Furthermore, we provide an analysis of the performance of this approach and ways to improve it. Our experimental results show an average slowdown of around $1.3\times$ when compared to binaries compiled for specific issue-widths. Although the "price" for this flexibility seems quite high, we show that with simple compiler optimizations this can be reduced to less than $1.09\times$. Unfortunately, we did not have access to the compiler to incorporate these changes, however, through extensive measurements we demonstrate that this penalty can be reduced if these optimizations are implemented. On the other hand, our approach can be exploited for many other purposes, such as: dynamic performance/energy trade-off and energy-saving mechanisms, dynamic hardware sharing (when executing multiple threads), and dynamic code insertion for hardware fault detection mechanisms.

The remainder of the paper is structured as follows. In Section II, we discuss other approaches to splitting instruction bundles into smaller bundles, executed over multiple cycles and other similar work. Subsequently, Section III provides several reasons for why the ability to split instruction bundles over multiple cycles is useful. In Section IV we explain our approach to supporting execution on different issue-widths. In Section V, we show the performance results of our approach, and we also propose certain optimizations and estimate what

[1] We refer to a generic binary when the binary can be executed by different issue VLIW processors.

978-3-9815370-0-0/DATE13/©2013 EDAA

the performance would be when using those. In Section VI, we discuss additional possible optimizations and other future work. Finally, in Section VII we summarize our results and present our conclusions.

## II. Related work

There have been some efforts to optimize the resource utilization of applications running on VLIW cores by adapting the issue-width. However, as far as we are aware, ours is the first to attempt to dynamically change the issue-width of a running application. In [6], the authors target a clustered VLIW processor that can disable individual clusters. They use a profiler to determine if a specific cluster is unused and if so insert instructions to disable that cluster. This allows them to gain performance, while limiting the power consumption caused by the additional clusters.

Similar to our work is Extended Split Issue [7], which is based on Split Issue [8]. Both approaches attempt to solve the problem of binary compatibility between different versions of a VLIW core. They do so by delaying the writes to the register file until all instructions have been executed. This allows the core to execute code compiled for a core with different instruction latencies. In Extended Split Issue, it is demonstrated that this technique can be used to execute instructions one at a time and in any order. This allows bundles to be split over multiple cycles to enable Simultaneous Multi-Threading [9] (SMT). The authors of [10] propose a modified version of the split issue approach which uses clustering to reduce the hardware cost of implementing split issue. This approach uses the fact that bundles intended for different clusters write to separate register files, to split the execution over multiple cycles. The drawback of the Split Issue approach is that it requires additional hardware in the form of buffers and queues to dynamically issue instructions [10]. Instead, by modifying the assembler and compiler, our approach requires only minimal hardware modifications. Because our approach does not rely on clustering we also avoid the overhead of copying data between register files of different VLIW clusters.

In [11] the authors develop a VLIW processor that can dynamically switch between 2-, 4-, and 8-issue. This work targets the same processor, with the intent of providing the ability to run the same application on different issue-widths using only a single binary. Before introducing our approach in Section IV, we motivate our approach in the following section.

## III. Motivation

In this paper, we propose a new approach in code generation for VLIW processors that can dynamically switch between different issue-widths without the need for a complex instruction scheduler (such as those found in out-of-order scalar RISC processors). More specifically, we propose an approach to generate a single executable (binary) that can be executed on a processor with different issue-widths. This approach has the following advantages:

- *single-thread advantages*: first, our approach allows the processor to switch between issue-widths at any point during execution without needing to introduce checkpoints. More importantly, when switching to a different issue core, there is no need for complex algorithms to ensure a thread is restarted at the same point in a different version of the application code. As such, a lot of hardware overhead during thread execution and design complexity is avoided. Second, different applications exhibit different characteristics, such as type of operations and instruction-level parallelism, but even within a single application, different phases can be detected that have distinct and different characteristics [12]. With our approach, we can exploit these characteristics to optimize for example resource utilization when the ILP is low by making them available for other threads or gating them off to save power. Third, we can dynamically trade-off performance with power consumption and our approach will allow for the introduction of new algorithms to determine optimal execution based on, for example, being powered by battery or the wall socket. Fourth, having a single binary for different processor organizations means that when the decision is made to switch, there is no longer the need to load a new binary for the new organization and thereby saving time and power (of the I-cache).
- *multiple thread advantages*: when executing multiple threads, such threads are likely to have priorities, translating into the need for interruptibility. With the provided flexibility, we no longer need to stop the execution of the running thread(s) in order to make “room” for the new thread, but instead we can dynamically continue all running threads on lower-issue cores and run the new thread on the newly freed resources.
- *advantages for fault-tolerance*: in software-based self-testing [13] (SBST) systems, test software needs to be inserted from time to time in order to test the hardware. This case is similar to running a higher-priority thread and will cause the running thread to be interrupted and execution halted until the test software has finished execution. Using our approach, we can simply assign less lanes for the running thread and run the test software on the freed lanes to test those lanes. In this manner, all the lanes can be tested by selecting different lanes to test at different times, without ever stopping the running threads.

It must be clear by now, that our approach can be used in many different scenarios with many advantages.

## IV. Approach

Our approach in creating a generic binary relies on the fact that when targeting a VLIW architecture, all instructions within a bundle are independent. This means that if we can avoid read after write (RAW) hazards and premature branches, we can execute instructions within a bundle in any order.

### A. Target Processor

For this approach we target an implementation of the VEX architecture [14] called $\rho$VEX [15], which is a parametrized VLIW processor. It allows us to specify the issue-width, and

TABLE I
LAYOUT OF FUNCTIONAL UNITS IN AN 8-ISSUE $\rho$VEX.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| ALU | ALU<br>MUL | ALU | ALU<br>MUL | ALU<br>MEM | ALU<br>MUL | ALU<br>BRANCH | ALU<br>MUL |

TABLE II
LAYOUT OF FUNCTIONAL UNITS IN A 4-ISSUE $\rho$VEX.

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| ALU<br>MEM | ALU<br>MUL | ALU<br>BRANCH | ALU<br>MUL |

the number and position of functional units. The target for the generic binaries is an 8-issue machine with 8 ALUs, 4 Multipliers, 1 Load/Store unit and 1 Branch unit. The layout of the functional units is shown in Table I. The layout for 4- and 2-issue versions that support the generic binary is shown in Table II and Table III respectively. When splitting a bundle, groups of instructions are executed from the larger bundle from left to right.

We use this layout because the larger configurations are made up of the smaller, 2-issue, configuration repeated several times. In the larger configurations the additional load/store and branch units are disabled. The reason for the position of the branch unit and load/store unit is explained in the next section.

### B. General approach

In order to generate code that can run on an 8-, 4-, and 2-issue processor, we have to guarantee that the results are the same, regardless of whether the bundle was split or not. Listing 1 shows an example of instructions that cannot be executed in any order without introducing hazards. When instruction number 5 is executed before instruction number 7, the results will be incorrect. Consequently, in order to ensure that a bundle of instructions can be split into smaller bundles, it must meet certain requirements.

Listing 1. Example of an instruction bundle for an 8-issue $\rho$VEX.

```
1    c0    shru $r0.13 = $r0.9, 24
2    c0    and $r0.11 = $r0.11, 63
3    c0    and $r0.14 = $r0.9, 63
4    c0    sh2add $r0.12 = $r0.12, (a)
5    c0    shru $r0.9 = $r0.9, 16
6    c0    and $r0.4 = $r0.4, 63
7    c0    shru $r0.15 = $r0.9, 8
```

The first requirement is that it must be possible to sort the instructions in such a way that they can be split over multiple cycles without introducing RAW hazards. This effectively means that if we construct a false dependency graph of the instructions, the graph must be acyclic. If the graph is cyclic, a RAW hazard cannot be avoided as shown in Listing 2.

TABLE III
LAYOUT OF FUNCTIONAL UNITS IN A 2-ISSUE $\rho$VEX.

| 0 | 1 |
|---|---|
| ALU<br>BRANCH<br>MEM | ALU<br>MUL |

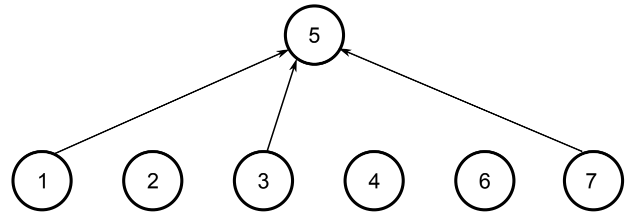

Fig. 1. Directed graph corresponding to the bundle shown in Listing 1.

Regardless of how the instructions are ordered, the result will be wrong when the bundle is split into smaller bundles. Instructions that read and write the same registers such as the one on line 6 in Listing 1 pose no problem and are still allowed.

Listing 2. Example of a bundle that cannot be split.

```
1    c0    shru $r0.13 = $r0.15, 24
2    c0    shru $r0.9 = $r0.13, 16
3    c0    shru $r0.15 = $r0.9, 8
```

The second requirement is that if a bundle contains a branch instruction, it cannot be executed before other instructions in the same bundle. This means that the branch instructions must always be placed at the end of the bundle. This has the additional effect of limiting the number of allowed false dependencies on branch instruction to one, otherwise the bundle cannot be correctly split. We solve this by not writing to branch registers being read by a branch instruction in the same bundle. This will require at most 1 additional branch register.

A similar problem occurs with load/store operations simply because the target organization only supports a single load/store unit. This means that the position of these instructions within the bundle is always fixed, which limits us when these instructions write to registers that are read by other instructions, or the other way around. This is why the position of the load/store unit is placed near the center of the bundle.

Lastly, because code scheduled for an 8-issue machine often contains many additional NOPs, running the same code on a 4- or 2-issue machine will cause overhead in the form of unneeded NOPs being executed. In order to alleviate these performance issues, we implemented a way for the hardware to detect the last useful instruction and have it skip many of the unnecessary NOPs.

### C. Assembler implementation

In the assembler, we construct a directed graph from the instructions based on the source and destination registers of each instruction. Each instruction is represented by a node, and each dependency is represented by an edge between the two nodes. An edge from node 1 to node 2 indicates that node 2 writes to a register read by node 1. Figure 1 depicts the graph created for the bundle shown in Listing 1. Nodes with no incoming edges have no false dependencies and can be sorted by the assembler.

The nodes are placed using a depth first search of all possible positions. Placing a node succeeds if the instruction slot is free, and contains the correct functional unit required

TABLE IV
A VALID SORTING OF THE NODES IN FIGURE 1.

| slot # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| node # | 1 | 2 | 3 | 6 | 4 | - | 7 | 5 |

for the instruction. After the instruction is placed in a slot any children of this node are updated with the position that this node is placed at. This ensures that the child nodes are placed after the parent node. For instance if node 1 is placed in slot 3, and node 2 is dependent on node 1, then node 2 has to be placed in slot 4 or higher. Table IV shows how the assembler sorts the nodes in Figure 1 into the different issue slots. Because instruction 4 has an address argument, it takes up two issue slots, which is why it is not placed in slot 3.

After a node is placed, it is removed from the graph and its children are updated, causing additional nodes to become ready for sorting. The sorting process is repeated until all the nodes have been placed. After all the nodes are sorted into execution slots, the assembler determines the last non-NOP instruction in the bundle, and sets the *Last Bit*. This bit is used by the hardware to determine whether or not to skip ahead to the next bundle.

### D. Hardware implementation

In order to support generic binaries in hardware we made two changes to our design [11]: The first change was to make all branch offsets multiples of 2-bundles, instead of the actual size of the machine. When running the binary on a 4- or 8-issue processor the correct offset is generated by discarding 2 or 1 of the least significant bits, respectively.

The second change is to allow the fetch stage to skip to the next bundle whenever the *Last Bit* is detected in an instruction. To achieve this, we calculate a second program counter in the fetch stage. The first is the normal program counter, which is equal to the address of the next 2-, 4- or 8-issue bundle. The second program counter is always equal to the address of the next 8-issue bundle. When the *Last Bit* is detected, the address of the next 8-issue bundle is used to fetch the next instruction instead of the program counter, effectively skipping unnecessary NOP instructions. Because both program counters are calculated in parallel, the impact on the cycle time is negligible.

## V. RESULTS

In order to test our approach we use the Powerstone embedded benchmark suite [16]. This benchmark suite consists of several stand-alone applications that are easily compiled without the need for porting additional libraries. Using these benchmarks we perform several different measurements. First, we compare native execution to generic binary execution on the different issue-widths. Finally, we also have the place and route results to show the impact of the hardware changes on the performance of the device.

### A. Experimental Setup

The results are obtained from execution traces generated using Modelsim [17] and a VHDL model of the $\rho$VEX processor. We instantiate 2-, 4-, and 8-issue versions of the core to run the different binaries. The simulation outputs execution traces for later analysis.

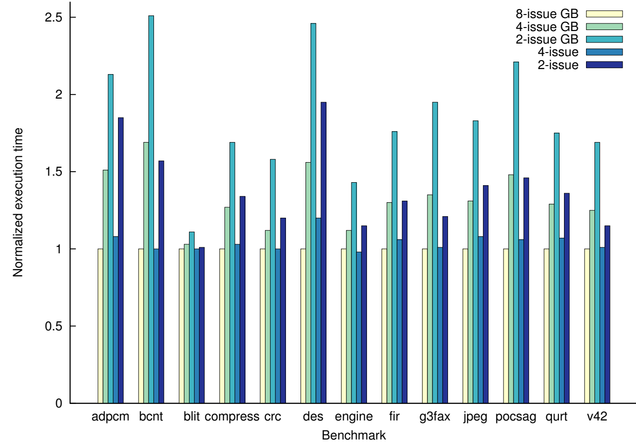

Fig. 2. The execution time for the different benchmarks normalized on the execution time of the 8-issue version.

The benchmarks are compiled using the HP VEX compiler [18] for 2-, 4-, and 8-issue. The 8-issue version is assembled as a generic binary using the techniques described in Section IV-C. The 2- and 4-issue versions are assembled normally to serve as a baseline comparison for the performance overhead.

Because the compiler we use is only available as a binary, we were not able to implement the requirements, mentioned earlier, into the compiler. Instead we implement a check in the assembler and manually fix the assembly as required.

### B. Generic vs Native

In order to determine the performance impact of the proposed approach we compile 3 different version of each benchmark as explained in Section V-A. The generic binary is then run on 2-, 4-, and 8-issue machines, and compared to the results of the native 2-, and 4-issue versions.

Figure 2 shows the execution time of each benchmark normalized on the execution time of the 8-issue version. This figure shows us that reducing the issue-width by half does not double the execution time, however, it also shows that for many of the benchmarks the native 2-issue and 4-issue versions are significantly faster than the version compiled as a generic binary.

The slowdown for each benchmark is shown in Figure 3. For the 4-issue, the slowdown ranges from 1.03 to 1.69 with an average slowdown of 1.27, while for the 2-issue the slowdown ranges from 1.10 to 1.61 with an average of 1.34.

When splitting instruction bundles, there can be two major sources of performance loss. The first is inefficient instruction packing, meaning that the bundle is split over more cycles than needed. The second inefficiency is introduced when the compiler produces a bundle of 5 instructions followed by a bundle of 3 instructions, when a bundle of 4 followed by 4 would have been possible. The result is that when the bundles are split it will result in an additional cycle compared to the

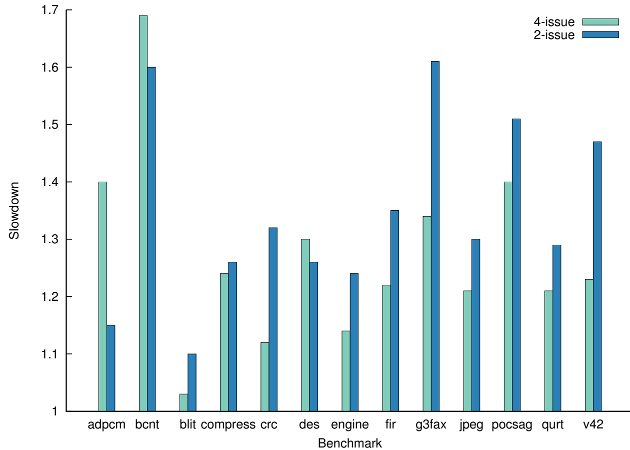


Fig. 3. The slowdown caused by the generic binary relative to a native binary.

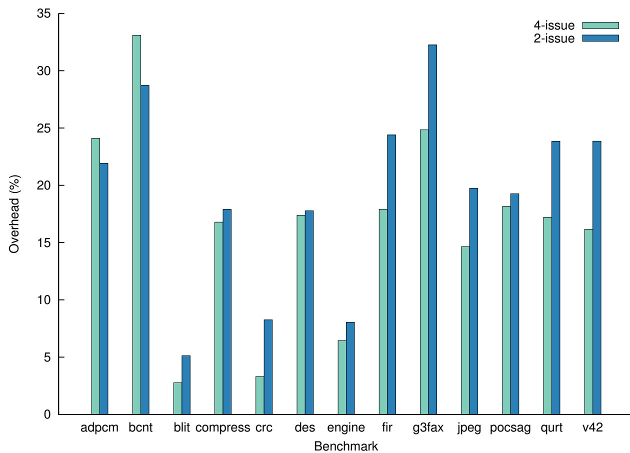


Fig. 4. Percentage of bundles that are caused by packing inefficiency.

native binary. In order to understand which of these is the case we analyze the instruction traces.

Using the 2-, and 4-issue instruction traces, we determine the number of bundles that would be executed under ideal conditions. Figure 4 shows the percentage of bundles caused by inefficient packing. This graph corresponds nicely to the graph of the slowdown for each application and shows that packing efficiency is the major source of overhead. After analyzing the badly packed bundles we determine that roughly 77% of the badly packed bundles involve load and store instructions in the case of a 4-issue configuration, and 57% in the case of a 2-issue configuration.

In order to understand why bundles containing load/store instructions are so badly packed we look back at the instruction layout. Because we are not able to modify the compiler, and load instructions can write to registers used by other instructions, we placed the load/store unit in slot 4. This allows us to assemble the program with as few manual changes to the assembly as possible. However, the result is that whenever a load/store instruction is combined with less than 3 other instructions, this will result in the instructions being packed inefficiently. A similar issue exists for branch instructions, however, load/store instructions are more common, and because of that, responsible for the majority of the inefficiency.

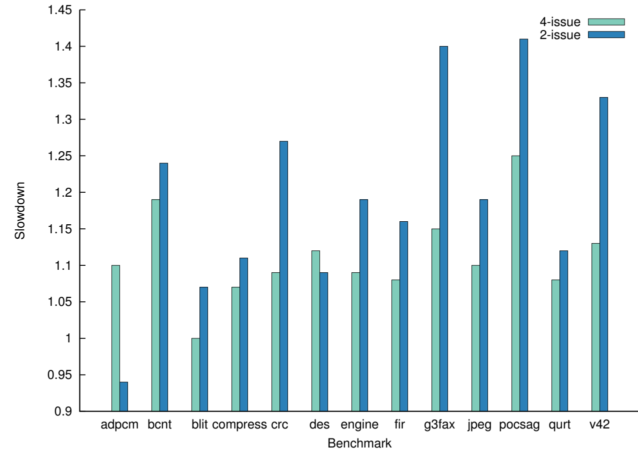


Fig. 5. Slowdown compared to baseline with improved packing for load/store instructions.

Because we now know the cause of the a large part of the execution overhead, we can estimate the performance of using an improved compiler that will avoid false dependencies involving load/store instructions. Using the data obtained from the instruction traces we can determine how much improvement we could obtain by simply subtracting the number of badly packed bundles caused by load/store operations from the total cycle count. This gives us a rough estimate of what the improved performance could be. The results are shown in Figure 5. With this improvement the average overhead for the 4-issue generic binary over a native 4-issue binary is estimated to be around 10%. The highest overhead is at 25% for 4-issue and 40% for 2-issue.

Strangely, we also notice in Figure 5 that for the 2-issue adpcm benchmark, with improved instruction packing taken into account, that the generic binary performs better than the native 2-issue version. We speculate that this is because the compiler is not optimized for low issue-widths.

Lastly, using these numbers we can also calculate that if we were to achieve perfect instruction packing, the overhead compared to native compilation would be on average 7% for a 4-issue configuration and 9% for a 2-issue configuration.

### C. Register Utilization

Because creating the generic binaries involved modifying the assembly by renaming certain registers in order to satisfy the requirements of the assembler, we determined the total register utilization over the lifetime of the application. In Table V, we show the number of registers that are never used over the entire lifetime of the application. This shows that in all but the adpcm benchmark a large number of free registers are available to perform the manual register renaming, without the need to introduce additional load/store operations.

This same information is also relevant to the earlier discussion about optimizing the instruction packing by modifying the compiler to avoid load/store instructions writing to registers read by other instructions in the same bundle. Since none of these benchmarks use the full number of registers, it

TABLE V
THE NUMBER OF UNUSED REGISTERS DURING THE EXECUTION OF EACH BENCHMARK.

| Benchmark | Unused registers |
|---|---|
| adpcm | 3 |
| bcnt | 45 |
| blit | 51 |
| compress | 42 |
| crc | 52 |
| des | 42 |
| engine | 38 |
| fir | 25 |
| g3fax | 50 |
| jpeg | 36 |
| pocsag | 25 |
| qurt | 26 |
| v42 | 35 |

TABLE VI
PLACE AND ROUTE RESULTS FOR DIFFERENT ISSUE-WIDTHS OF $\rho$VEX, WITH AND WITHOUT MODIFICATIONS

| Issue-width | Resource | Original | Modified |
|---|---|---|---|
| 2 | LUTs | 3506 | 3486 |
| | Regs | 728 | 669 |
| | MHz | 143 | 143 |
| 4 | LUTs | 8110 | 8137 |
| | Regs | 1185 | 1192 |
| | MHz | 137 | 143 |
| 8 | LUTs | 23571 | |
| | Regs | 2199 | |
| | MHz | 100 | |

should be possible for the compiler to schedule load and store instructions *without additional overhead*.

### D. Place and route results

Table VI shows the place and route results for 2-, 4-, and 8-issue cores with and without support for generic binaries. The table shows that the impact on area and clock frequency is quite small. In the case of the 2-issue core, the modified version is actually slightly smaller, while the modified 4-issue core achieves a higher clock frequency than the unmodified one. The change in clock frequency is due to restrictions placed on how long immediates are handled in the case of the generic binary. This restriction leads to simplified decoding logic in the 4-issue core, resulting in a slight improvement in clock frequency.

## VI. Future work

In order to improve the performance of the generic binaries, we propose to modify the the compiler by adding the requirement that load instruction cannot write to registers that are read by other instructions in the same bundle. These modifications will be done to a GCC port for the $\rho$VEX processor. A similar modification will ensure that there are no instructions writing to branch registers used by a branch instruction in the same bundle. This will eliminate the need for modifying assembly files by hand as we have done in these experiments.

## VII. Conclusion

In this paper, we introduced a new approach in code generation for VLIW to allow for dynamic issue-width adaptation and highlighted several advantages for different operational scenarios. Even though our approach incurs a performance hit, we must note that in some of the mentioned scenarios, our approach will allow for continued execution of the running application (thereby making up for lost performance) and remove the need for multiple code versions that would greatly penalize performance when switching codes. Specifically, this approach allows us to switch between issue-widths at run-time without the need for check-points.

Our results show that our implementation suffers on average a 30% overhead when compared to a natively compiled application. Further analysis indicates that this can be reduced to 10% or lower with simple compiler modifications. We also show that the overhead in area consumption is negligible, and in some case even beneficial.

## References


[1] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: a Technology Platform for Customizable VLIW Embedded Processing," in *Proceedings of the 27th Annual International Symposium on Computer architecture*, 2000, pp. 203–213.

[2] J. T. J. Van Eijndhoven and E. J. D. Pol, "TriMedia CPU64 Architecture," in *Proceedings of the 1999 IEEE International Conference on Computer Design*, 1999, pp. 586–.

[3] H. Sharangpani and K. Arora, "Itanium Processor Microarchitecture," *IEEE Micro*, pp. 24–43, Sep. 2000.

[4] G. K. Yeap, *Practical Low Power Digital VLSI Design*, Aug 1997.

[5] G. Keramidas, V. Spiliopoulos, and S. Kaxiras, "Interval-based Models for Run-time DVFS Orchestration in Superscalar Processors," in *Proceedings of the 7th ACM international conference on Computing Frontiers*, 2010, pp. 287–296.

[6] B. V. Iyer, J. G. Beu, and T. M. Conte, "Length Adaptive Processors: A Solution for the Energy/Performance Dilemma in Embedded Systems," in *Interact-13: Workshop on Interaction Between Compilers and Computer Architecture (Held in Conjunction with HPCA)*, 2009.

[7] B. Iyer, S. Srinivasan, and B. Jacob, "Extended Split-Issue: Enabling Flexibility in the Hardware Implementation of NUAL VLIW DSPs," in *Proceedings of the 31st Annual International Symposium on Computer architecture*. IEEE Computer Society, 2004, pp. 364–.

[8] B. R. Rau, "Dynamically scheduled VLIW processors," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, 1993, pp. 80–92.

[9] D. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *In 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392–403.

[10] M. Gupta, F. Sanchez, and J. Llosa, "A low cost split-issue technique to improve performance of SMT clustered VLIW processors," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –12.

[11] F. Anjam, M. Nadeem, and S. Wong, "Targeting Code Diversity with Run-time Adjustable Issue-slots in a Chip Multiprocessor," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2011)*, March 2011.

[12] Z. Yu, N. Puzovic, A. Portero, and R. Giorgi, "Characterizing Phase Behavior for Dynamically Reconfigurable Architectures," in *HiPEAC ACACES-2011*, jul 2011, pp. 89–92.

[13] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Reorda, "Microprocessor Software-Based Self-Testing," *Design Test of Computers, IEEE*, no. 3, pp. 4 –19, may-june 2010.

[14] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. 500 Sansome Street, Suite 400, San Francisco, CA 94111: Morgan Kaufmann Publishers, 2005.

[15] S. Wong, T. van As, and G. Brown, "$\rho$-VEX: A Reconfigurable and Extensible Softcore VLIW Processor," in *International Conference on Field-Programmable Technology (ICFPT)*, December 2008.

[16] http://www.cprover.org/goto-cc/examples/index.php.

[17] http://model.com.

[18] http://www.hpl.hp.com/downloads/vex/.