

# Architecture for Runtime Hardware Compilation

Geoffrey Ndu and Jim Garside

School of Computer Science, The University of Manchester, Oxford Road,  
Manchester, M13 9PL, UK,  
{gndu, jgarside}@cs.man.ac.uk

**Abstract.** Runtime compilation of a platform-independent intermediate representation may overcome issues of programmability and portability in reconfigurable processors, allowing them become mainstream. Dynamic compilation makes the hardware transparent to the programmer and improves the portability of applications. However compilation overheads, which are significantly greater than in software compilation, must be surmounted. Targeting simple hardware, compiling only frequently executing code and saving compilations are employed to reduce compilation overheads making dynamic compilation effective. This paper describes a feasibility study on the effectiveness of this approach to hardware dynamic compilation and shows advantages when a small proportion of the code is retained as a hardware implementation.

## 1 Introduction

Reconfigurable processing offers high-performance and energy efficiency to a variety of applications [16, 5]. However, Reconfigurable Processors (RPs) are not common on today's computing devices because of programming and portability issues. Mapping an algorithm to a programmable logic usually involves special languages or tools, reducing its appeal to programmers. Logic on RPs usually do not share architectures, even on platforms from the same family, making porting from one device to another difficult and expensive.

Dynamic compilation [3] — the runtime compilation of platform independent intermediate representation to native code — is now common on computing systems because of the prevalence of high-level language Virtual Machines (VMs) [18] such as Java [15]. Many of today's platform independent intermediate representations retain enough semantic information that hardware mapping could be performed at runtime making the reconfigurable logic transparent to programmers as well as improving the portability of applications. However, hardware dynamic compilation is not yet common as time-consuming tasks such as place and route, are performed at runtime making it difficult to amortise compilation costs.

This paper explores a new dynamic compilation system that can reduce and hide hardware compilation costs making runtime compilation viable on RPs. Its design is based on three key ideas: (1) targeting coarse grained reconfigurable logic thereby reducing the time and resource requirements for mapping

algorithms to hardware, (2) limiting compilation to only hot functions to allow the amortisation of compilation costs better and (3) saving compilations in persistence manner enabling the recovery of overheads over multiple runs of an application. A successful feasibility study was conducted on the viability of the system using a proof-of-concept prototype. The rest of the paper describes the system, presents performance results and outlines future work.

## 2 Related Work

Reconfigurable computing [7] has been studied extensively and its benefits are well recognised. However, the difficulty of programming such systems and the limited portability of their binaries are still obstacles to their widespread adoption. Warp-processing [16] employed dynamic binary translation [8], to tackle programming and code portability issues. It translates binary code sequences to hardware transparently – requiring no extra designer effort nor causing disruption to the standard software development tool flow – by profiling an executing binary program, detecting critical regions, decompiling them, synthesising them to hardware, placing and routing them onto a custom on-chip FPGA, and updating the binary to call the hardware next time. Its CAD algorithms, which run on a separate microprocessor, require significant resources as well as time to execute. The use of an FPGA limits it to a few loops because of the large memory required to save FPGA configurations and, consequently, to applications where a few loops dominate.

Dynamic Instruction Merging (DIM) [5] also dynamically and transparently translates software to hardware. The translator is hardware implemented and is designed for a coarse-grained architecture. Each microprocessor instruction is translated as it is fetched and sequences are saved in a cache indexed by the Program Counter (PC). Next time the same sequence is fetched the translated sequence is loaded onto the programmable logic from the cache and executed as an atomic unit. The processor’s PC is then updated to allow execution to continue in software.

Binaries usually don’t contain enough high-level information for extensive optimisations, limiting the repertoire of optimisations available with warp-processing and DIM. Decompilation [6], the recovery of high-level information from a low-level software representation such as a binary or assembly code, could be employed, but it is expensive and does not always recover enough high level information for extensive optimisation.

Neither of these systems save and reuse translations across executions. It was mentioned that warp-processing can operate in saved configuration mode for short running applications but the implementation was not discussed. The practicalities of maintaining translation code caches across execution has been investigated [17].

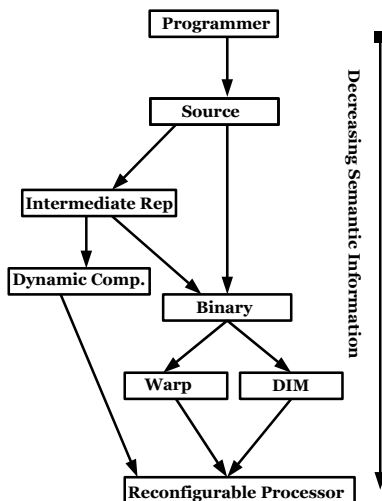


Fig. 1: Approaches to Transparent &amp; Dynamic Partitioning

### 3 Background

The main difference between the approach described here and earlier systems is the use of an intermediate code format instead of decompiling a binary (see Fig 1). This can contain rich program information that isn't expressed in a binary, enabling more sophisticated compiler analyses and transformations. It also enables 'write-once run-everywhere' programming paradigm on RPs.

The substrate of our dynamic compilation system (VM and dynamic compiler) is the Low Level Virtual Machine (LLVM)[14] *lli*. The LLVM instruction set (bitcode) is RISC-like with key high-level information such as type information, explicit control flow graphs and an explicit dataflow representation. Consequently, the same code representation is used for all the stages of compilation, eliminating the need to translate the virtual instruction set to a compiler intermediate representation before applying optimisations. This leads to a smaller compilation overhead. LLVM is language independent with a transparent runtime model like microprocessor binary. It doesn't require complex runtime services such as class loading and can be used to implement low level system routines.

*lli* executes applications in LLVM bitcode format using a compile only strategy. It optimises all functions to the same degree, rather than focusing on frequently executed functions, thus potentially introducing some unnecessary compilation overhead. A lazy compilation technique, relying on callback functions, is used to allow functions to be compiled only when needed.

Traditionally VMs/dynamic compilers are usually designed to fit an existing implementation of a particular architecture, consequently opportunities to blend symbiotically software and hardware are often missed. We co-designed our software and hardware for improved performance and design simplicity. Our hardware is a Reconfigurable Instruction Set Processor (RISP) [4], a subset of RPs where the programmable logic is a standard functional unit of the proces-

sor, as they have the least communication overhead compared to other types of RPs.

## 4 System Architecture

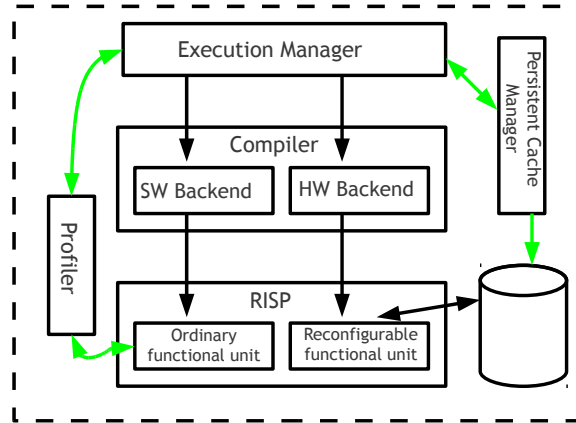


Fig. 2: System Architecture

The architecture of our VM and how it interacts with the RISP is depicted in Fig. 2. The Execution Manager (EM) starts off execution by using the compiler lazily to translate functions (the compilation process is shown in Fig. 3) from LLVM to native code. It then activates the profiler and notifies it of the function(s) it is interested in by providing their start and end addresses. Whenever a function exceeds a certain execution threshold, the profiler notifies the EM (see section 5.1). The EM then recompiles the function to hardware. Recompilation is achieved by generating for each basic-block (or part of a large basic-block) microinstructions, from the LLVM code representation, to configure the programmable logic. The compiler also emits native code anew but now each group of instructions implementing a basic-block is replaced with an opcode which points to the location of the hardware configuration for the basic-block. Essentially, opcode(s) is synthesised for each basic-block on the fly. On subsequent execution each opcode invokes the hardware configuration it represents (see section 5) which then runs on the reconfigurable unit as an ‘atomic’ unit.

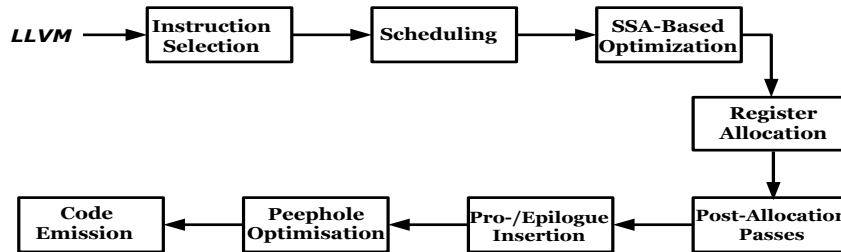


Fig. 3: Compilation Phases

To allow the recovery of overheads over multiple runs the VM saves all recompiled functions in a persistent manner across application invocations and reuses them when they are encountered again. A possible design of such a persistent code cache is presented in section 4.1.

#### 4.1 Persistent Cache Management

Our persistent cache is a file in a non-volatile memory below the main memory. It contains hardware configurations, native code used to invoke those configurations and associated translation data structures. The Persistent Cache Manager (PCM) is responsible for generating and managing persistent caches. It maintains a database of all available caches and provides mechanisms for efficient lookup of caches.

The PCM requests, from the operating system, a linear region of memory from the virtual address space for use as persistent memory pool when dynamic compilation is invoked. Hardware configurations are now generated and run from memory allocated from the persistent memory pool. Information is written to file whenever the PCM has no free memory to allocate or an application performs the exit system call.

On start-up, the EM also performs a cache lookup through the PCM. If the application already has a persistent cache, functions may be recovered from this rather than being recompiled.

#### 4.2 Hardware Dynamic Compiler

Compiling for a Coarse-Grained Reconfigurable Architecture (CGRA) is similar to VLIW [9] compilation, Software pipelining [13] is usually employed to exploit both loop-level and instruction-level parallelism, the key difference being in scheduling. Scheduling on CGRAs requires operand values to be routed explicitly between producing and consuming operations while on VLIWs routing is implicitly guaranteed by storing intermediate values in a multi-ported, centralised register file. Scheduling is further complicated by the ability of most CGRAs to use functional units as computation or routing resource. We are currently exploring the suitability of various software pipelining technique for our compiler. In the short term, we have developed a naive and simple postpass scheduler leveraging the LLVM compiler infrastructure [14].

Our current scheduler is greedy, it schedules operations at the earliest possible time once all the input operands are available and a functional unit is available to execute it on.

## 5 Hardware Architecture

Our hardware is the ARM926EJ-S[2], a simple, in-order, 5-stage pipelined processor, enhanced it with a Reconfigurable Functional Unit (RFU) consisting of

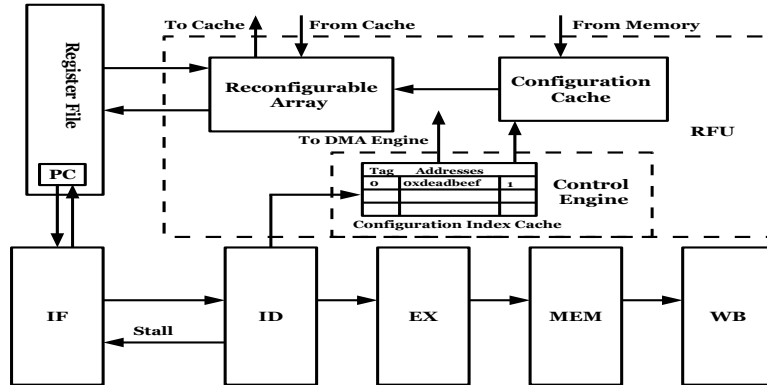


Fig. 4: Microarchitecture of RISP

a Control Engine (CE), a coarse-grained Reconfigurable Array (RA) and a Configuration Cache (CA). The RFU is just another functional unit and has access to the register file. Our microarchitecture and architecture are depicted in Fig. 4 and 5 respectively. An instruction, *lrc*, was added to the processor's instruction set to allow the compiler delineate patterns for execution on the RFU. The immediate field in *lrc* is used to specify a unique identifier for each hardware configuration in an application.

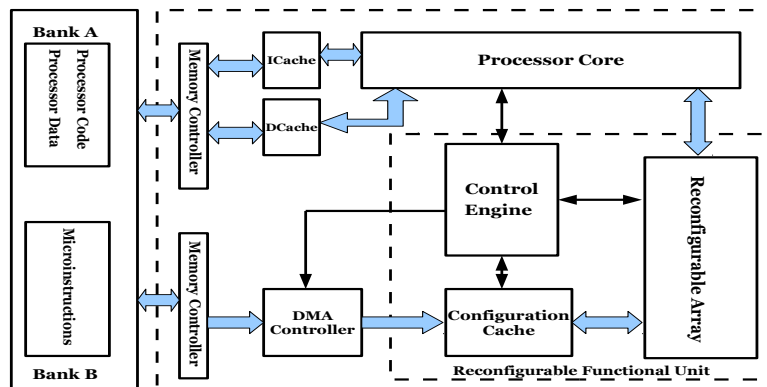


Fig. 5: Architecture of RISP

The RFU is autonomous and only depends on the processor for decoding instructions and stalling the pipeline when the RFU is in use. The CE is responsible for loading the correct configuration into the Configuration Cache (CC) with aid of the Configuration Index Cache (CIC) and *lrc*'s immediate field. The CIC is an ordinary cache, each line contains the memory addresses of a hardware configuration and a bit representing the presence of the configuration in the CC. If a configuration is absent in the CC, the CE stalls execution while it fetches the configuration from memory using the Dynamic Memory Access (DMA) engine. The CE initiates execution by pointing its sequencing engine to the first microinstruction of the configuration. The sequencer now loads a new microinstruction (this reconfigures the RA) from the CC into the RA, on each cycle

until it encounters a termination microinstruction. The CE then updates the PC to allow execution to continue in software. The CIC entries are generated by the compiler/VM and are loaded using ARM926EJ-S coprocessor instructions. Logically our RA is organised as shown in Fig 6. Each functional unit has two

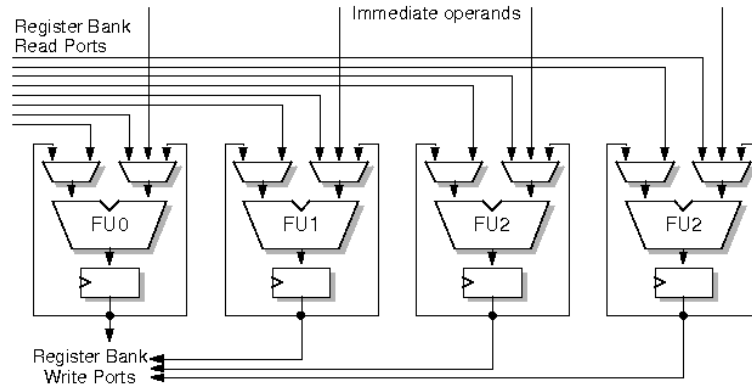


Fig. 6: Logical Organisation of the RA[12]

registers, results of the stage of execution can be written into one, both or neither of these registers. A one bit global register is used to enable the predication of execution (not shown in Fig 6). Each of the functional unit can read any of the registers in the other functional units (not shown in Fig 6) allowing the RA to match the shape of the data flow graph it is executing but at the of cost more configuration bits and more wire for routing.

### 5.1 Profiler

The profiler is an independent hardware unit within the processor. It operates on the principle that a function  $f$  is ‘hot’ if its execution count  $e$ , sampled every  $n$  cycles, exceeds a threshold  $t$  at the end of an interval of  $m$  cycles.  $e$ ,  $f$ ,  $m$ ,  $n$ ,  $t$  can be varied by the EM, allowing it not only to profile specific functions while filtering out those elements that are not of immediate interest but to determine how detailed profiling will be.

The EM initiates profiling by passing function start and end addresses to the profiler using ARM coprocessor instructions. The profiler employs a simple cache like structure to keep track of functions and their execution counts. It increments the execution counter associated with a function if it determines that the value of the PC lies within that function. At the end of period of  $m$  cycles, all functions with execution counts greater than a threshold,  $t$  are reported to the EM as candidates for recompilation.

## 6 Evaluation and Performance

We developed a virtual prototype — a fully functional software representation of hardware encompassing processors and peripherals — of the RISP described

in section 5, based on OVPsim [1], to obtain a rough first estimate of performance. It is based on a cycle approximate model of a ARM926EJS Integrator Compact Platform and is detailed enough to execute operating systems allowing us to develop the software components of the VM as if on real hardware. In our prototype, RA reconfiguration takes 0 cycles while latency of operations are 1 and 2 processor equivalent cycles for ALU (on all functional units) and simple multiplications (on 1 functional unit) respectively. The RFU performs load and stores in 1 processor equivalent cycle (on 2 functional units). The CC only fetches configurations on demand and replaces them using the least recently used technique.

We conducted experiments whose objectives were to : (1) get a rough estimate of the costs of hardware compilation for different applications, (2) determine if those costs could be recovered within a reasonable time (20 invocations of each application with a different dataset each time) using our system, and (3) compare the performance our VM to LLVM *lli*. Applications and datasets are from MiBench [11] and MIDataSets [10] respectively. To make the measurement of compilation overheads easier the VMs are single-threaded.

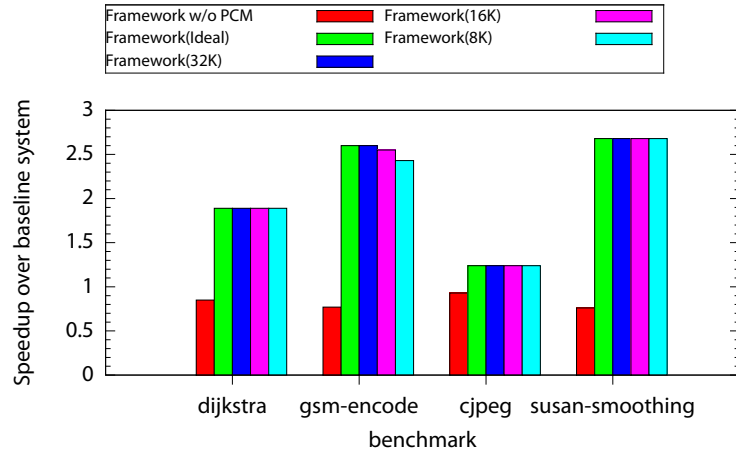


Fig. 7: Performance Estimates

We compared the execution time of each application with *lli* to: (1) execution time with our VM but without the PCM (*Framework w/o pcm*), (2) execution time on an ideal version of our VM i.e. zero CC overhead (*Framework ideal*) and (3) execution time on our VM with CC of 8K,16K and 32K (*Framework 8K - 32K*). Results, presented in Fig. 7, show that compiling frequently used code sequences to hardware brings no benefits because of overheads unless those compilations are saved and reused. Further, quite a small CC is sufficient to sustain performance in our system.

Significant energy savings can only be achieved in reconfigurable processors if most of the processing is mapped into hardware whilst the mapping overhead is kept low. Table 1 shows the maximum number of functions compiled to hardware and their contribution to the execution time; compilation overheads are

Benchmark	No of Recompiled Functions	Execution Time (%)
dijkstra	2	78
gsm-encode	2	94
cjpeg	4	93
susan-smothing	1	99

Table 1: Recompiled Functions Statistics

not included. However, recompiling critical code sequences may not necessarily translate to improved energy efficiency since compilation of cold code sequences and other VM services executing in software contribute to the energy profile. An application which runs for a short time relative to the time spent in the VM may see no reduction in its energy usage even with all its critical code sequences compiled to hardware. Therefore, an efficient VM is a necessity for effective dynamic compilation.

## 7 Conclusions & Future Work

This paper presented a hardware dynamic compilation system which operates on the principle that compiling frequently executing functions to CGRAs plus saving and reusing those recompilations can make runtime hardware compilation effective. We described the software and hardware components of our VM and presented preliminary results from experiments on our virtual prototype. Dynamic compilation system plays a significant role in determining the overall energy savings made by a dynamically compiled application so further reducing the system’s overhead is important.

We are about to study desktop-based applications to determine if our system can still be effective with such applications. The few applications evaluated thus far are typical embedded applications — a few functions dominate execution time — making it less difficult to amortise compilation cost but desktop applications tend to have their execution spread over more regions. Furthermore, much of the promise of such a system is in energy efficiency and a detailed hardware model to obtain better power estimates is needed.

**Acknowledgements** The authors would like to thank Imperas Software Limited for supporting this research through tool provision.

## References

1. Open Virtual Platforms<sup>TM</sup>(OVP<sup>TM</sup>). <http://www.ovpworld.org>
2. ARM926EJ-S Technical Reference Manual. ARM Ltd (2008)
3. Aycock, J.: A brief history of just-in-time. *ACM Comput. Surv.* 35(2), 97–113 (2003)
4. Barat, F., Lauwereins, R.: Reconfigurable instruction set processors: a survey. In: *Proceedings. 11th International Workshop on Rapid System Prototyping*. pp. 168–173 (2000)

5. Beck, A., Rutzig, M., Gaydadjiev, G., Carro, L.: Transparent reconfigurable acceleration for heterogeneous embedded applications. In: Proceedings. Design, Automation and Test in Europe. pp. 1208–1213 (2008)
6. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. *Softw. Pract. Exper.* 25(7), 811–829 (1995)
7. Compton, K., Hauck, S.: Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.* 34(2), 171–210 (2002)
8. Ebcioglu, K., Altman, E.R.: DAISY: Dynamic compilation for 100% architectural compatibility. In: Proceedings. 24th International Symposium on Computer Architecture. pp. 26–37. ACM, New York (1997)
9. Fisher, J.A.: Very Long Instruction Word architectures and the ELI-512. In: Proceeding. 10th International Symposium on Computer Architecture. pp. 140–150. ACM, New York (1983)
10. Fursin, G., Cavazos, J., O’Boyle, M., Temam, O.: MiDataSets: Creating the conditions for a more realistic evaluation of Iterative optimization. In: Proceeding. 2nd International Conference on High performance Embedded Architectures and Compilers. pp. 245–260. Springer-Verlag, Heidelberg (2007)
11. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: Proceedings. 4th IEEE International Workshop Workload Characterization. pp. 3–14. IEEE Computer Society, Washington-DC (2001)
12. Ian, J.: Run-Time Object Code Compilation to Hardware. Ph.D. thesis, School of Computer Science, University of Manchester (2008)
13. Lam, M.: Software pipelining: an effective scheduling technique for VLIW machines. *SIGPLAN Not.* 23(7), 318–328 (1988)
14. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis transformation. In: Proceedings. 2nd International Symposium on Code Generation and Optimization. pp. 75–86 (2004)
15. Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
16. Lysecky, R., Stitt, G., Vahid, F.: Warp processors. *ACM Trans. Des. Autom. Electron. Syst.* 11(3), 659–681 (2006)
17. Reddi, V.J., Connors, D., Cohn, R.S.: Persistence in dynamic code transformation systems. *SIGARCH Comput. Archit. News* 33(5), 69–74 (2005)
18. Smith, J., Nair, R.: Virtual machines: Versatile platforms for systems and processes. Morgan Kaufmann, San Francisco, USA (2003)