Boosting single thread performance in mobile processors via reconfigurable acceleration

DOI:
10.1007/978-3-642-28365-9_10

Boosting Single Thread Performance in Mobile Processors via Reconfigurable Acceleration

Geoffrey Ndu and Jim Garside

The APT Group
School of Computer Science,
The University of Manchester,
Oxford Road, Manchester.
United Kingdom
{g.ndu, jdg}@cs.man.ac.uk
http://apt.cs.man.ac.uk

Abstract. Mobile processors, a subclass of embedded processors, are increasingly employing multicore designs to improve performance. This often requires sacrificing resources in each CPU, degrading single thread performance which is still important according to Amdahl's law. The traditional technique for efficiently boosting serial performance in embedded processors, dedicated hardware acceleration, is unsuitable for modern mobile processors because of the heterogeneity and the diversity of applications they run. This paper proposes 'general purpose' accelerators, reconfigured on an application-by-application basis, as a means of increasing single thread performance. These accelerators are placed within the datapath of CPUs and support dynamic compilation. This paper presents the design of an architecture with such accelerators and evaluates the cost/performance implications of the design.

Keywords: reconfigurable, dynamic compilation, multicore, accelerator, JIT

1 Introduction

Mobile processors, a subclass of embedded processors, are General Purpose Processors (GPPs) designed primarily for small, fan-less, battery powered, mobile computing devices such as smart-phones. They are characterised by high performance, low energy consumption, small area and low cost. Mobile processors are increasingly moving to multicore designs to improve performance.

Multicore processors, multiple Central Processing Units (CPUs) on a die, improve performance by handling more work in parallel. Increasing the number of CPUs often requires sacrificing resources in each CPU which degrades single thread performance. Single thread performance is still important as some key applications have limited Thread-level Parallelism (TLP). Further, according to Amdahl's law [6], serial sections within a massively parallel application with lots of TLP are performance constraints. A current (and future) challenge for mobile processors vendors is how to efficiently increase single thread performance in these resource-constrained processors.

Unfortunately, the time-tested approach for serial performance improvement in embedded processors – accelerating compute intensive parts of applications using dedicated hardware – is unsuitable for modern mobile processors because of the heterogeneity and diversity of applications. The next best alternative is having 'general purpose' hardware, reconfigured on an application-by-application basis to realise frequently occurring functions. This is less efficient in terms of area, cost and power than fixed hardware but allows a GPP to be specialised based on the application it's currently running.

Reconfigurable hardware has been used successfully to accelerate single threads in experimental and commercial processors. However, employing it in multicore mobile processors poses two unique challenges.

A typical Reconfigurable Architecture (RA) is composed of several memory elements, programmable interconnect and an array of many Processing Elements (PEs) making its deployment prohibitive due to its significant area and power consumption. Further, mobile processors rely extensively on dynamic compilation, which is not yet common on RAs, to improve portability. Dynamic compilation is important as an increasing number of parallel programming systems rely on it to provide forward scaling [13]: applications that effectively scale with new core counts as well as the unavoidable augmentation and evolution of the instruction set. For instance, kernels (critical parallel functions) in Intel ® Array Building Blocks (IABB) [13] are first compiled to a platform independent Intermediate Representation (IR) then dynamically compiled to binary via a Virtual Machine (VM) at run-time.

This paper presents the VIrtual REconfigurable Micro-ENgine for Translation (VIREMENT), a mobile multicore processor employing general purpose accelerators to improve single thread performance. The general purpose accelerator is a Reconfigurable Functional Unit (RFU) placed within the datapath of each CPU. VIREMENT supports dynamic compilation by providing a run-time library for generating reconfigurable instructions on-the-fly. Experiments show an average performance improvement of 133% (2.33×) with area overhead of 34% per CPU.

2 Related work

Over the years, architectures that dynamically translate code to run on reconfigurable hardware have been developed. Such architectures eliminate dependencies on hardware features, letting hardware vendors significantly change features from one hardware generation to the next without breaking binary portability.

Warp [17] is a family of processors that automatically extracts and compiles kernels to Field-Programmable Gate Array (FPGA). A typical Warp processor is a System on Chip (SoC) with a main processor for executing applications, a less powerful processor on which a lean FPGA compiler runs, a profiler and a custom FPGA. It translates binary sequences to hardware transparently by profiling 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. However, 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 and consequently to applications where a few loops dominate because of the large memory required to save FPGA configurations.

The Configurable Compute Array (CCA) [11] is a matrix of simple, coarse-grained functional units coupled to a host CPU. Accelerating applications on the CCA involves two steps: the discovery/delineation of suitable, critical sub-graphs within the Directed Flow Graphs (DFGs) for the CCA and the replacement of such sub-graphs with micro-operations that configure the CCA. Static and dynamic approaches for sub-graphs selection were presented. Dynamic discovery involves a trace cache and its associated hardware optimiser which is rare in mobile processors because of cost and energy issues. Static discovery finds suitable code sequences for mapping onto the CCA at compile time using traditional compiler-based techniques for instruction set customisation.

CCA offered performance improvements to a variety of applications but no area, power nor latency measurements were provided making it difficult to evaluate the overall effectiveness of the approach. Further, CCA does not support shifts nor memory operations and handles only four inputs and two outputs thus limiting its application.

Dynamic Instruction Merging (DIM) [7] dynamically translates binaries to coarse-grained hardware using a hardware based translator. Translation is simultaneous with instruction fetch and translated sequences are cached on-chip. The next time a cached sequence is fetched the saved translation is executed atomically and the processor's Program Counter (PC) updated to allow software execution to continue. Custom Reconfigurable Arrays for Multiprocessor System (CReAMS) [18] is based on DIM but has a pipelined translator.

The translation algorithm is simple and fast as it is implemented in hardware but opportunities for optimising the many micro-operations produced by the translation process are missed. Further, energy is spent translating cold instruction sequences that contribute little to performance as DIM attempts to translate all instructions.

Another project [19] is based on a heterogeneous multicore processor where, cores being either Reconfigurable Hardware Unit (RHU)-cores or Reconfiguration Instruction Generation (RIG)-cores. An RHU-core is a superscalar CPU augmented with an RFU. The RIG-core is based on the same CPU as the RHU-core but with a hardware reconfigurable instruction generator. Each RIG-core services a number of RHU-cores and has no reconfigurable fabric. Each RHU-core collects traces of committed instructions and dispatches them to a RIG-core for translation. When the configuration is returned, it is stored in the RHU-core thread's address space. When next the start of a trace is detected the associated configuration(s) is fetched, decoded and processed by the RFU instead.

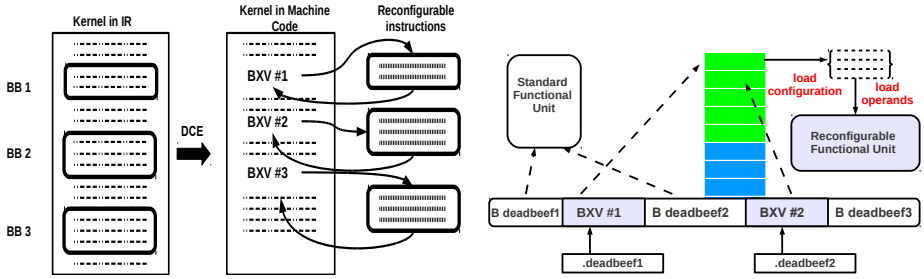
Despite its performance improvements this architecture may be too 'complex' for mobile processors as it uses trace caches with the consequent cost and energy penalties.

3 System Architecture

VIREMENT could be described as a multicore dynamic Application Specific Instruction-set Processor (ASIP) [14] that uses reconfigurable functional unit(s) instead of custom functional unit(s) to reduce power consumption and boost processing speed. It consists of a host CPU extended with reconfigurable hardware. Non-critical parts of an application are implemented using the standard instruction set (and run on standard

functional units) while kernels are implemented using a reconfigurable instruction set (micro-ops) which run on the reconfigurable functional unit(s). VIREMENT, unlike traditional stream oriented systems, takes small amounts of data at a time from the host's register file and produces another small amount of output. This imposes fewer restrictions on the characteristics of the application, allowing acceleration in most cases.

VIREMENT provides a compiler, the Dynamic Compilation Engine (DCE), to support dynamic compilation in VMs. In addition to its use in traditional 'write-once-run-anywhere' language VMs dynamic compilation is increasingly being employed in parallel programing systems to allow applications to 'forward scale' [13] as well as to support dynamic mapping [16]. DCE, based on Low Level Virtual Machine (LLVM) [15], provides dynamic code generation on VIREMENT and C/C++ APIs to enable integration into VMs. The code generation process is quite simple and is illustrated at a high level in Figure 1a. Initially, kernels in an application are identified and translated into a suitable IR. Each basic block from the critical functions is then translated into micro-ops by the DCE. The original basic block is extracted from the function and replaced by a single instruction which serves as a pointer to the memory location of the micro-ops for that particular basic block. Each of the original basic block, now replaced by a special instruction pointing to its configuration, executes as an atomic unit on the reconfigurable functional unit(s) (Figure 1b). Essentially the DCE synthesises an application specific instruction, on-the-fly, to replace each basic block in a kernel. Large basic blocks may map into more than one group of micro-ops.



(a) Code generation on DCE (b) Reconfigurable execution on VIREMENT

Fig. 1: Overview of compilation and execution on VIREMENT

4 Microarchitecture

VIREMENT is simply a 4-core processor with each CPU augmented with an RFU. Logically each core can be divided into the CPU and the VIREMENT Reconfigurable Functional Unit (VRFU).

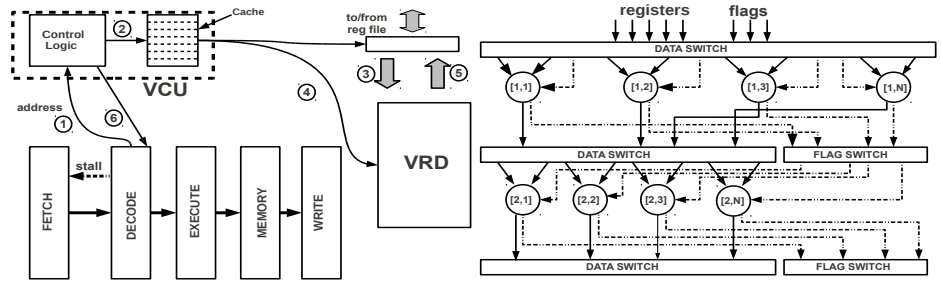
4.1 CPU

The CPU is comparable to the ARM926EJ-S and runs at 200 MHz. It has a simple, in-order, 5-stage pipeline, Harvard architecture, RISC core which is replicated four times.

Each VIREMENT core has separate L1 data and instruction caches. The data cache is two-way banked allowing two simultaneous cache accesses per cycle if the two cache accesses are to different banks

Each CPU supports three types of instruction sets: ARM, Thumb and VIREMENT Execution Environment (VEE). ARM is the 32-bit main instruction set while Thumb is the 16-bit subset of ARM. VEE is the reconfigurable instruction set, micro-ops, but can only be accessed by executing a special 'ARM' instruction "Branch-to-Virement"(BXV). Decoding micro-ops is the responsibility of the VRFU. Therefore, each BXV serves only as a pointer to a single context of micro-ops, uniquely identified by the address encoded in the BXV.

Figure 2a shows how the RFU is integrated into the CPU. The decode stage is modified to stall the pipeline once it recognises a BXV. The decoder simply forwards the address portion of the BXV to the VRFU and awaits for the completion signal from VRFU to remove the stall signal. The VRFU itself consists of an array of PEs, the VIREMENT Reconfigurable Datapath (VRD) and the VIREMENT Control Unit (VCU). The VCU is mainly responsible for managing reconfiguration.



(a) Instruction execution on VRFU(numbers denote sequence)

(b) ALU interconnection on theVRD

Fig. 2: Overview of compilation and execution on VIREMENT

4.2 VIREMENT Reconfigurable Datapath

The VRD comprises an array of interconnected PEs with data routed using multiplexers. It is purely combinational to reduce its complexity, latency and energy consumption. The PEs are arranged into rows, with each row connected to the next through a switch box. Computation flows from top to bottom with each switch box capable of connecting any of the previous row output to any of the next row PE input. This restrictive interconnection simplifies P&R facilitating dynamic compilation on the RFU.

The fundamental processing element in the VRD is the Arithmetic and Logic Unit (ALU). The ALUs are simple to reduce latency, cost and energy consumption. Currently only integer operations are supported. Each operation has three operands, two 32-bit values and a 1-bit flag. Operations generate a four flag bits: a *sign flag*, a *zero flag*, an *overflow flag* and a *carry out flag*. These are similar to the host CPU's allowing the core and the VRFU to exchange flags. Figure 2b shows how the ALUs are interconnected.

In addition to ALU operations, a few PEs per row are capable of loads/stores through the multi-banked L1 data cache. The address of each load/store must be calculated in an ALU in the previous row. All data accesses are through the L1 cache and the VCU stalls the VRD in the case of a miss. Channelling all memory accesses through the L1 cache allows the VRFU to out-source cache coherency management to the CPU. This keeps the VRFU design and programming simple.

4.3 VIREMENT Control Unit

The VCU's primary responsibility is to manage reconfiguration. It has a small SRAM for caching configuration contexts fetched from the main memory. Contexts are supplied almost instantaneously to VRD from the local cache. The VCU fetches configuration contexts directly from the main memory, via the DMA. Configuration contexts are all the same size making cache management simple and eliminating cache fragmentation. The cache employs a LRU replacement policy.

5 Structure of the Dynamic Compiler Engine (DCE)

The DCE generates code on the fly for the VRFU, starting from LLVM IR [15]. As with time- and resource-limited run-time compilers used on mobile processors, emphasis is on speed, small memory footprint and energy efficiency rather than code quality. The DCE relies heavily on the LLVM compiler framework [15] for transformations and analysis.

Dynamic compilation incurs substantial overhead. Further, quality may suffer in the quest to generate code within a limited time budget. However, overheads are largely amortised in the typical DCE usage model: compiling relatively small, critical sections in a long running application. Quality issues could be tackled with split compilation, performing time-consuming analysis offline and saving the results for run-time use.

LLVM's Static Single Assignment (SSA) IR offers a number of advantages to DCE. It can serve both as a persistent, offline code representation and as a compiler internal representation, with no semantic conversions needed between the two [15]. It is increasingly being used in parallel compilation systems targetted by DCE. For instance, AMD embeds LLVM IR source for kernels in its OpenCL Binary Image Format (BIF) 2.0 [5].

5.1 Code Generation Process

The generation of reconfigurable instruction is basically a post-pass optimisation within the CPU code generator. CPU instructions are first generated and then translated into micro-ops. This allows for the seamless intermixing of standard and reconfigurable instructions since not all operations can be performed on the VRFU. The pass is fast and lean (it is slightly more complex than the code generators in software binary translators) allowing its use in mobile devices with constrained processing power and storage. However, it has advantages over present hardware reconfigurable translators as more sophisticated optimisations and post-fabrication modifications are enabled.

Code generation can be logically divided into nine distinct steps which are:

1. **DAG Formation:** The first step is the expansion of the LLVM input into a Direct Acyclic Graph (DAG) of LLVM instructions.
2. **Instruction Selection:** This step converts the DAG of LLVM instructions into a DAG of native CPU instructions using a pattern-matching instruction selector.
3. **Scheduling and Formation:** In this pass, a scheduler assigns a linear order to the DAG from the previous stage. The DAG is now converted to a sequential list of *MachineInstr*s [4] and destroyed. *MachineInstr* is an abstract way of representing machine instructions in LLVM. It represents a machine instruction as an op-code number and a set of operands.
4. **Register Allocation & SSA Deconstruction:** Virtual registers are eliminated from instructions and replaced with physical registers.
5. **Reconfigurable Instruction Generation:** This is the pass that extracts and converts CPU instructions into micro-ops with each set of extracted CPU instructions being replaced by a single BXV instruction. The pass is a functional level pass i.e. it executes on each function in the program independent of all of the other functions.
   (a) **Instruction Translation:** This stage identifies and translates supported *MachineInstr*s to micro-ops. Here, instructions are extracted, sequentially, from the list of *MachineInstr*s and translated into micro-ops represented using VIREMENT Intermediate Representation (VIR). A micro-op in VIR is an n-tuple consisting of an operator and operands. Each instruction is given a unique number, called an *ID*, as it is translated. *ID*s help in tracking dependencies between micro-ops.
   Translation starts from the beginning of a basic block and ends when an unsupported instruction or the end of the basic block is encountered. Listing 1 shows an example translation. If the number of translated *MachineInstr*s is above a certain threshold compilation proceeds to item 5b else the translated micro-ops are discarded and translation restarts at the next instruction beyond the unsupported one (see pseudo code in algorithm 1).
   (b) **Micro-ops optimisation:** A number of optimisations could be applied to the micro-ops at this stage. Presently, the main one is the removal of copy instructions. Copy (register-to-register move) instructions, are redundant on VRFU as operands can be moved directly from producers to consumers.
   (c) **Micro-ops Placement and Routing:** This stage involves the simultaneous placement and routing of micro-ops on the VRFU. The output of this stage is pseudo-assembly code for the VRFU. Placement and Routing (PR) of the micro-ops uses a simple, single-pass greedy algorithm (subsection P & R Algorithm) to keep resource consumption and time overhead to a minimum. The algorithm simply takes a micro-op from the linear VIR and determines, based on data dependencies and resource availability, where to place it on the VRD.
   (d) **Micro-ops Code Emission:** Binary code is generated for the VRFU along with 'glue' code needed for loading operands and writing results back to the CPU. This happens during code emission for the CPU.
6. **Code Emission:** The completed machine code is emitted into memory ready for execution. Each BXV in the machine code points to a corresponding configuration.

```
Input: MBB /* Basic block of MachineInstrs from step 4 */
Output: MBB_BXV /* Basic block of MachineInstrs with BXV instructions */
Output: Configs_1,2,...,n /* VIREMENT configurations */
1  while MachineInstr in MBB do
2      Translation_Buffer ← initialise_translation_buffer(void);
3      while MachineInstr is supported do
4          Micro-op ← translate_to_microop(MachineInstr);
5          save_microop_in_translation_buffer(Microops);
6      end
7      if number_translated_MachineInstr < threshold then continue;
8      Translation_Buffer_opt ← optimize_microops(Translation_Buffer);
9      Translation_Buffer_pr ← place_&_route(Translation_Buffer_opt);
10     if P_&_R fails —— P_&_R unbeneficial then continue;
11     MBB_BXV ← replace_successfully_routed_machineinstr_with_bxv(MBB);
12     Configs_1,2,...,n ← emit_microop_to_memory(Translation_Buffer_pr);
13 end
```

Algorithm 1: Pseudo code for the Reconfigurable Instruction Generation pass

P & R Algorithm The algorithm is quite simple: the first step is to retrieve the next unscheduled micro-op in the VIR. The operands (including flags) are read to verify dependencies. Dependencies are tracked using a small data structure called the *Dependency table* which shows the row and column on the VRD where each operand was last defined. The columns are numbered from left to right while the rows are numbered from top to bottom. The row numbers of all the source operands are compared and the operand with the highest row number determines where the micro-op is to be placed.

The next step is to search for a free PE on the VRD to place the micro-op in. Resource usage is modelled with a matrix-like structure, *PE Table*, which has the same dimensions as the VRD. Each element represents a PE and contains information such as resource availability. Each row in the *PE Table* is scanned from left to right, starting from the row determined by the *Dependency table*, until a free unit is found. The *Dependency table* is then updated if the micro-op just place defines a value(s). The configuration for the multiplexers are generated from information stored in the *PE Table*.

If the VRFU size is 4x4 and the first micro-op from Listing 1.2 is already placed in PE_{00}. A query to the *Dependency table* for the second instruction will return row 1 as *%r5* and *%f1* are defined (have entries in the *Dependency table*) by the first instruction in row 0. The other source operands do not have entries in the *Dependency table* and need to be fetched from the register file, so they have no influence on the placement. The *PE Table* is then scanned from left to right starting from row 1. PE_{01} is empty so the micro-op is placed on it and the *Dependency table* table updated to reflect that *%r3* is now defined by PE_{10}. The next use of *%r3* (third instruction) must now be placed in a row higher than 1.

Listing 1 Example translation MachineInstrs to micro-ops. The subscript numbers in Listing 1.2 are the IDs. *%f,%i,%r,%t* denote flag, register,immediate and temporary operands.*%f1* means that flag is supplied by instruction with ID 1 and *%i8* is an immediate of size 8 bits. The PC relative branch in Listing 1.1 stops translation.

```
bb12:
%r5 = adds %r4,%r3
%r3 = adc %r2,%r5
%r4 = ldr [%r3,-%r0]
br %i8
```

Listing 1.1 Translation: MachineInstrs

```
%r5 = add1 %r4,%r3
%r3 = adc2 %r2,%r5,%f1
%t1 = sub3 %r3,%r0
%r4 = ldr4 [%t1]
```

Listing 1.2 Translation: Micro-ops

6 Evaluation

6.1 Performance Evaluation

To evaluate the performance the architecture a cycle approximate simulation model of VIREMENT based on GEM5 [9] was developed. The parameters used in the model were derived by describing VIREMENT in Verilog and synthesising the description using the Synposys DC compiler [3] with the Nangate 45 nm cell library [2]. The initial development of the DCE was done on Open Virtual Platform (OVP) CPIntegratorPlatform [1] (a virtual platform). The simulator runs under Linux with a memory of 256 MB and no swap space. This is similar to the execution environment in a typical modern smart-phone. The size of the VRD is 4X4 with two PEs per row capable of performing loads and stores using address calculated in the previous row.

Program	Benchmark Suite	Application Domain	Parallelization Model	Implementation	Total Instructions (Billions)
fib	BOTS	integer	tasks	OpenMP	7.02
sort	BOTS	integer sorting	tasks	OpenMP	6.34
bfs	Rodinia	graph	tasks	OpenMP	16.69
freqmine	Parsec	data mining	data-parallel	OpenMP	43.90
nqueens	BOTS	games	tasks	OpenMP	61.31

Table 1: Description of benchmarks

Benchmarks from Parsec [8], Rodinia [10] and Bots [12] were compared (the benchmark suite is described in Table 1) running on VIREMENT and using DCE for code generation against statically complied versions running on a baseline. The baseline is exactly like VIREMENT less the VRFU. The benchmarks are largely integer benchmarks as VRFU does not yet support floating point operations. We modified benchmarks for VIREMENT to mimic parallel systems that dynamically compile kernels by compiling kernels to LLVM IR and embedding them in the native binary. The compilation of kernels to LLVM IR does not require special preparation and was done with a standard compiler. The host code was then modified to trigger a DCE based VM when a kernel was called for the first time. The VM's input was LLVM IR with code cache for storing generated code. We only measured performance (speedup) over the parallel region(s) of each benchmark.

Figure 3 shows speedup, the ratio of the execution time on baseline to execution time on VIREMENT to for each benchmark described in Table 1. *nqueens* gained most running on VIREMENT with a speedup of 2.7 against *sort* with only 1.4. On average, VIREMENT is faster than the baseline by 2.3×. This is largely attributable to the increased Instruction Level Parallelism (ILP) offered by VRD. This, combined with the relatively small overhead of DCE (on average about 2% of the execution time on VIREMENT) and the small number of compilations, only a few functions need to be compiled for VIREMENT to outperform the baseline.

The benchmarks that experienced significant boost in performance, such as *nqueens* and *freqmine* all have dominant kernels with significant ILP. However, ILP is limited within basic blocks especially non-numeric programs. This suggests that enabling DCE to compile across basic block boundaries would improve performance further. Presently, we are enhancing DCE to support PR across basic block boundaries. We are also developing a pass that allows the DCE code generator to quickly estimate the benefits of compiling a piece of code. This will save energy and time compared to the present approach where DCE has to translate and PR code before finding out if the code sequence will benefit from running on VIREMENT.

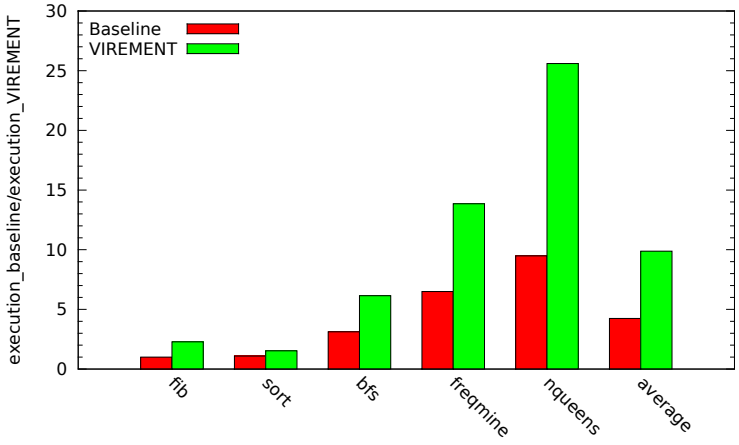


Fig. 3: Results of performance evaluation

Figure 4 shows the percentage of execution time spent compiling and the number of kernels (critical functions) complied by DCE for each application. Table 2 shows for each benchmark the number of LLVM instructions complied and the compilation overhead in cycles.

6.2 Area Evaluation

Estimates from synthesis show that VRFU is only 34% of each VIREMENT CPU because of simplicity of the design, (Table 3). The addition of the VRFU increases the size

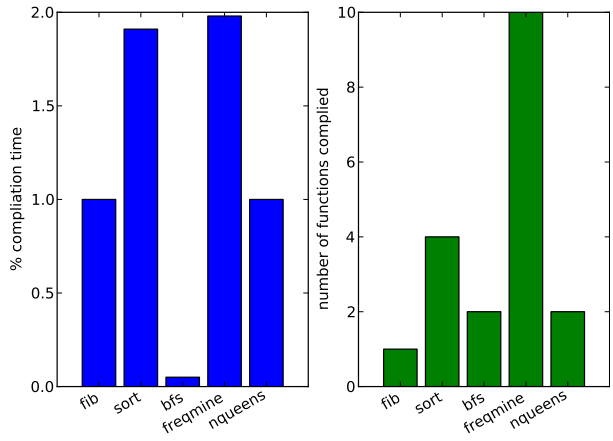


Fig. 4: Compilation Statics

of the CPU by half with majority of the area of the VRFU occupied by the VRD. We believe that the size of the VRFU could be reduced significantly with design and synthesis optimizations. Assuming each gate is a NAND gate (4 transistors) each VRFU requires about 473,000 transistors.

Program	**Num. LLVM Instructions**	**Compilation Overhead (Millions Cycles)**
fib	9	31.16
sort	324	58.27
bfs	280	52.37
nqueens	89	49.27
freqmine	2194	352.86

Table 2: Compilation Overhead

Component	Gate Equivalents
CPU	226,000
VRD	66,808
VCU	51,612

Table 3: Area breakdown

7 Conclusions

This work demonstrates that it is possible to use a reconfigurable hardware to improve single thread performance on resource constrained mobile processors in a cost effective manner. It also showed how dynamic compilation could be provided on such an architecture. We obtained mean speedup of up 2.33× over five diverse programs while increasing the area of each CPU by only 52%.

8 Acknowledgements

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

References

1. Open Virtual Platform™, http://www.ovpworld.org
2. Si2., http://www.si2.org
3. Synopsys Inc., http://www.synopsys.com
4. The LLVM Target-Independent Code Generator, http://llvm.org/docs/CodeGenerator.html
5. AMD Accelerated Parallel Processing OpenCL®. Advanced Micro Devices, Inc., Sunnyvale, CA, USA. (August 2011), http://developer.amd.com/sdks/amdappsdk/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf
6. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proc. of the April 18-20, 1967, spring joint Comp. conf. pp. 483–485. AFIPS '67 (Spring), ACM, New York, USA (1967)
7. Beck, A.C.S. et al.: Transparent reconfigurable acceleration for heterogeneous embedded applications. In: Proc. of the Conf. on Design, Automation and Test in Europe. pp. 1208–1213. DATE '08, ACM, New York, USA (2008)
8. Bienia, C. et al.: The PARSEC benchmark suite: characterization and architectural implications. In: Proc. of the 17th Int. Conf. on Parallel Arch. and Compilation Techniques. pp. 72–81. PACT '08, ACM, New York, USA (2008)
9. Binkert, N.L. et al.: The M5 simulator: Modeling networked systems. IEEE Micro 26, 52–60 (July 2006)
10. Che, S. et al.: Rodinia: A benchmark suite for heterogeneous computing. In: Proc. of the 2009 IEEE Int. Symp. on Workload Characterization (IISWC). pp. 44–54. IISWC '09, IEEE Comp. Society, Washington, USA (2009)
11. Clark, N et al.: Processor acceleration through automated instruction set customization. In: Proc. of the 36th annual IEEE/ACM Int. Symp. on Microarchitecture. pp. 129–. MICRO 36, IEEE Comp. Society, Washington, USA (2003)
12. Duran, A. et al.: Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Proc. of the 2009 Int. Conf. on Parallel Processing. pp. 124–131. ICPP '09, IEEE Comp. Society, Washington, USA (2009)
13. Ghuloum, A. et al.: Future-Proof Data Parallel Algorithms and Software on Intel™ for Multi-Core Architecture. Intel Technology Journal 11(4), 333 –347 (Nov 2007)
14. Keutzer, K. et al.: From ASIC to ASIP: the next design discontinuity. In: Comp. Design: VLSI in Comp.s and Processors, 2002. Proc.. 2002 IEEE Int. Conf. on. pp. 84 – 90 (2002)
15. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the Int. Symp. on Code Generation and Optimization. pp. 75–86. CGO '04, IEEE Comp. Society, Washington, USA (2004)
16. Luk, C.K. et al.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proc. of the 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture. pp. 45–55. MICRO 42, ACM, New York, USA (2009)
17. Lysecky, R. et al.: Warp processors. In: Proc. of the 41st annual Design Automation Conf. pp. 659–681. DAC '04, ACM, New York, NY, USA (2004)
18. Rutzig, M.B. et al.: CReAMS: an embedded multiprocessor platform. In: Proc. of the 7th Int. Conf. on Reconfigurable computing: architectures, tools and applications. pp. 118–124. ARC'11, Springer-Verlag, Berlin, Heidelberg (2011)
19. Suri, T., Aggarwal, A.: Improving scalability and per-core performance in multi-cores through resource sharing and reconfiguration. In: VLSI Design, 2009 22nd Int. Conf. on. pp. 145 –150 (2009)