Compiling for Automatically Generated Instruction Set Extensions

Alastair Murray
a.c.murray@ed.ac.uk

Björn Franke
bfranke@inf.ed.ac.uk

Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom

ABSTRACT

The automatic generation of instruction set extensions (ISEs) to provide application-specific acceleration for embedded processors has been a productive area of research in recent years. The use of automatic algorithms, however, results in instructions that are radically different from those found in conventional ISAs. This has resulted in a gap between the hardware's capabilities and the compiler's ability to exploit them. This paper proposes an innovative high-level compiler pass that uses subgraph isomorphism checking to exploit these complex instructions. Our extended code generator also enables the reuse of ISEs designed for one application in another, which may be a newer version of the same application or a different one from the same domain. Operating in a separate pass permits computationally expensive techniques to be applied that are uniquely suited for mapping complex instructions, but unsuitable for conventional instruction selection. We demonstrate that this targeted use of an expensive algorithm effectively controls overall compilation time. The existing, mature, compiler back-end can then handle the remainder of the compilation. Instructions are automatically produced for 179 benchmarks, resulting in a total of 1965 unique instructions. The high-level pass integrated into the open-source GCC compiler is able to use the instructions produced for each benchmark to obtain an average speed-up of 1.26 for the ENCORE extensible processor.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation*

General Terms

Design, experimentation, performance

Keywords

Code generation, instruction set extensions, subgraph isomorphism

1. INTRODUCTION

Specializing processors for a particular domain is an effective way of increasing the performance achievable for a given level of power consumption. The most obvious examples of this are the different processor families for different domain areas. *Digital Signal Processors* (DSPs) are based on VLIW or static superscalar designs, with scratchpad memories and specialized instructions (e.g. a multiply-accumulate, MAC, that is common in DSP tasks). Other processor families include Network Processors, or general purpose embedded processors ranging from microcontrollers, used in everything from hard-disk drives to cars to washing machines, up to high performance processors used in portable media players, smartphones, netbooks, etc.

While selecting the correct processor from the correct family and then configuring it appropriately results in a processor well-matched to a task, designing custom hardware can dramatically reduce the power required for the targeted task [12]. Taking this idea to its full extent results in *Application Specific Integrated Circuits* (ASICs), which are very high performance and low power, but take time and effort to develop so they are neither low cost nor have a short time-to-market. Additionally, once they have been deployed their functionality is set, new features may not be implemented and bugs cannot be fixed. An increasingly popular compromise between ASICs and general purpose embedded processors are *Application Specific Instruction-set Processors* (ASIPs). These processors take a pre-verified baseline processor as a core and add extension instructions, thus standard tasks can use the baseline processor but critical kernels can be programmed to use the extension hardware. This strikes a balance between performance and time-to-market, and the pre-verified baseline avoids much of the risk involved in developing new hardware.

Manually designed ASIPs can outperform general purpose embedded processors, use less power and are quicker and cheaper to design than ASICs [13]. Using *Automated Instruction Set Extension* (AISE) to automatically design ASIPs, however, improves on this yet again. A set of automated tools can do this by profiling the application to find hot-spots and analyzing the data-flow at these spots to produce instruction definitions. These definitions can then be used to automatically create the hardware based on the data-flow graphs [5]. Examples of such extensible processors are the SYNOPSYS ARC 600 and 700 series, the TENSILICA XTENSA, the ARM OPTIMODE and the MIPS PRO series.

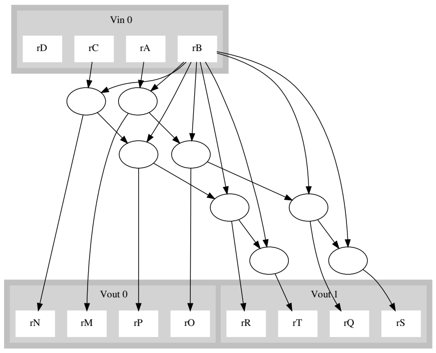
Using AISE to design ASIPs is an effective way to design hardware [11]. The problem, however, is that the capabilities of compilers lag behind the features of the hardware. This paper, therefore, investigates how the compiler can effectively use an AISE produced processor.

The standard methodology for using extension instructions within programs is for the AISE tool to note where it finds each exten-

```
  data = input;
  coef = coefficient;
  sum = 0.0;
  for (i = 0; i < 8; i++) {
#     term1 = *data++;
#     term2 = *coef++;
      sum += term1 * term2;
  }
  *output = sum;

.
```

(a) Snippet from UTDSP *lmsfir 8-1 ptrs*.



(b) Extension instruction generated from the code in figure (a).

```
  data = input;
  coef = coefficient;
  sum = 0.0;
  term1 = *data++;
  term2 = *coef++;
  for (i = 1; i < 8; i++) {
      sum += term1 * term2;
#     term1 = *data++;
#     term2 = *coef++;
  }
  sum += term1 * term2;
  *output = sum;

.
```

(c) Snippet from UTDSP *lmsfir 8-1 ptrs SWP*.

Figure 1: The extension instruction in figure (b) was generated from the code in figure (a). It is possible, however, to map it to both the code in figure (a) and (c). The lines of code that it implements are marked by a #, it executes four iterations of those lines in one instruction as GCC unrolls both loops by a factor of four. Figure (c) is the *software pipelined* version of figure (a).

sion, but there is no compiler support for the generated ISEs. This is due to the directed acyclic graph (DAG) structure and complexity of the ISEs, which typically operate on more than two operands (up to 12 in our case) and generate more than one result (up to 8). Lack of compiler support is acceptable in the situation where a single program is being accelerated, but it creates an issue if the program needs to be changed after the processor has been fabricated. Re-running the AISE tool may generate different instructions requiring an engineer to manually map the old extensions to the new code, which is both time-consuming and error-prone [12]. It would be more appropriate for the compiler to automatically perform the mapping for the engineer. This, however, turns out to be a difficult task. The *Instruction Set Extensions* (ISEs) produced by AISE are often far too complicated for conventional tree-based instruction selection (indeed, the instructions are often DAGs, not trees, see figure 1(b)), and they are often far too large for peephole-based instruction selection. Some form of graph-based instruction mapping is required instead; this has been investigated previously [16, 10, 19], but for much smaller instructions than those produced by AISE.

1.1 Motivating Example

Figure 1 shows short snippets of codes from the UTDSP *lmsfir 8-1 ptrs* benchmark (figure 1(a)) and a version of the program which has been *software pipelined* (figure 1(c)). Figure 1(b) shows an extension instruction that was generated for the code in figure 1(a). It represents two sequential sequences of pointer incrementation (the two sequences can be calculated in parallel). The long sequences are possible because the GCC optimizer unrolled the loops by a factor of four. In a conventional AISE setup it would not be possible to re-use the extension instruction in a different piece of code because (a) rerunning the AISE tool on different code would result in different instructions being generated, or (b) the graph-based form of the extension instructions (see figure 1(b)) can not be processed by conventional tree-based instruction selectors. The technique presented in this paper was able to re-use the extension instruction, despite the different code-shape. This is because the subgraph still exists within the different shape so graph-subgraph isomorphism is able to identify where the instruction can be used. The lines marked with a # represent where the extension instruction was able to be used, in both in original file it was generated for, and the *software pipelined* version.

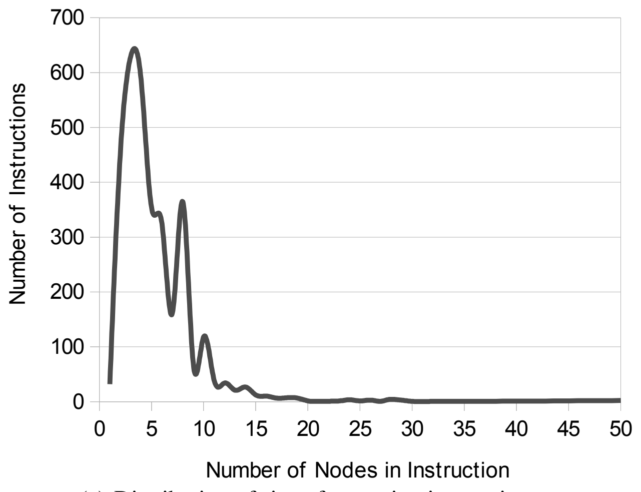
Most existing techniques for complex instruction mapping are designed for finding good, i.e. near optimal, solutions when using small graph-shaped instructions. They do not, however, generally scale well to very large instructions – most papers perform evaluations using instructions with only two operations in them. E.g. the approach taken in [9] finds every possible overlapping use of each instruction before selecting which ones it wants to use. This only works for small instructions, e.g. to map a single instruction (of two, three or eight nodes) in a single basic block of one hundred nodes has a worst-case complexity of $\binom{30}{2} = 435$, or $\binom{30}{3} = 4,060$ but $\binom{30}{8} = 5,852,926$. Although in reality it is unlikely that the worst case will occur, large instructions are still clearly not practical with this class of technique. Figure 2 shows the distribution of the number of nodes in instructions and basic blocks across all 179 benchmarks used for evaluation.

1.2 Contributions

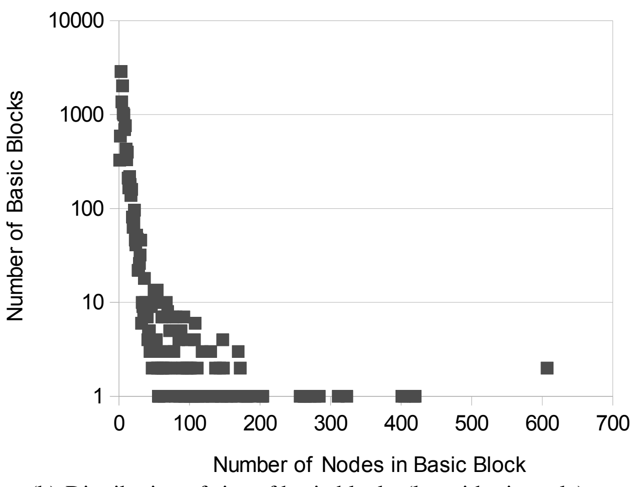
This paper makes following contributions:

1. We develop a method for mapping arbitrary graph-shaped instructions (both disjoint and not) to arbitrary programs. The novel aspect of the mapper is that it performs instruction mapping in the middle-end instead of the back-end. This allows it to focus only on extension instructions, that the back-end cannot exploit. The mature and tuned back-end performs effective instruction selection for the code that is not mapped to extension instructions.

2. We perform an extensive evaluation using 179 benchmarks from seven benchmark suites obtained from five sources. Results are generated using a hardware-verified cycle-accurate simulator. Additionally, the instruction mapper is evaluated using extension instructions generated for programs that are similar (but not identical to) the programs that they are mapped to, demonstrating the compiler's capability to re-use ISEs in programs other than the one they have been generated for.



(a) Distribution of size of extension instructions.



(b) Distribution of size of basic blocks (logarithmic scale).

Figure 2: The distribution of the number of nodes in the graphs used for graph-subgraph isomorphism.

1.3 Overview

The remainder of this paper is structured as follows. In section 2 we introduce extensible processors and the existing approaches to automated instruction set extension. This is followed in section 3 by a presentation of our novel code generation methodology for AISE generated instruction patterns. We demonstrate the effectiveness of our approach through experimental evaluation in section 4 before we discuss related work in section 5. Finally, in section 6 we summarize our results, conclude and provide an outlook to future work.

2. BACKGROUND

This section provides a short overview of the technologies relevant to the work of this paper.

2.1 Extensible Processors

Extensible processors are based on the premise that processor performance, die area, and power consumption can be improved if the architecture of the processor is extended to include some features that are application-specific. This approach requires an ability to extend the architecture and its implementation, as well as the compiler and associated binary utilities, to support the application-specific extensions.

Architecture extensions begin with the capability to add custom instructions to a baseline instruction set. In their simplest form these may be predefined packs of add-on instructions, such as the ARM DSP-enhanced extensions included in the ARM9E [3], the various flavors of MIPS Application Specific Extensions [17], or SYNOPSYS' floating-point extensions to the ARCOMPACT instruction set [2].

These are *domain-specific* extensions, they can be used across many related tasks. *Application-specific* instruction set extensions are not predefined by the processor vendor but are instead identified by the system integrator through analysis of the application. To allow such instructions to be incorporated into a pre-existing processor pipeline, there must be a well-defined extension interface. From a high-level architecture perspective this interface will allow the extension to operate as a "black-box" functional unit at the execute (*Ex*) stage of a standard RISC pipeline. This is an over-simplification though, standard RISC instructions are two-input and one-output. Effective extension instructions require this constraint to be relaxed as extensions exploit the parallelism available in large instructions. This, therefore, generally requires an extended or additional register file, hence the need for an extension interface.

Practical extensible processors for the embedded computing market, such as those from SYNOPSYS and TENSILICA, normally have single-issue in-order pipelines of 5-7 stages. This permits operating frequencies in the range 400-700MHz at the 90nm technology node. Extension instructions may be constrained to fit within a single clock cycle, or may be pipelined to operate across multiple cycles.

The representation of instruction set extensions varies from one vendor to another, but essentially describes the encoding and semantics of each extension instruction in ways that can be understood by both a processor generator tool and all of the software tools (e.g. compilers, assemblers and simulators). There follows a process of translating the abstract representation of the extension instructions to structural form using a Hardware Description Language (HDL) such as VERILOG or VHDL. This is then incorporated into the overall HDL definition of the processor, that is then synthesized to the target silicon technology or perhaps to an FPGA.

2.2 Automated Instruction Set Extension

Many algorithms for AISE have been described in the literature, [11] provides a comprehensive survey of the topic. The algorithm used for the generation of ISEs in following sections of this paper, however, is ISEGEN [5].

The most basic constraints on extension instructions are:

1. The template is *convex* (i.e. there is no dataflow path between two operations in the template that includes an operation that is not in the template), so that it may be scheduled.
2. Input and output *port constraints* are met (i.e. the number of register input and output ports are sufficient), so that it may

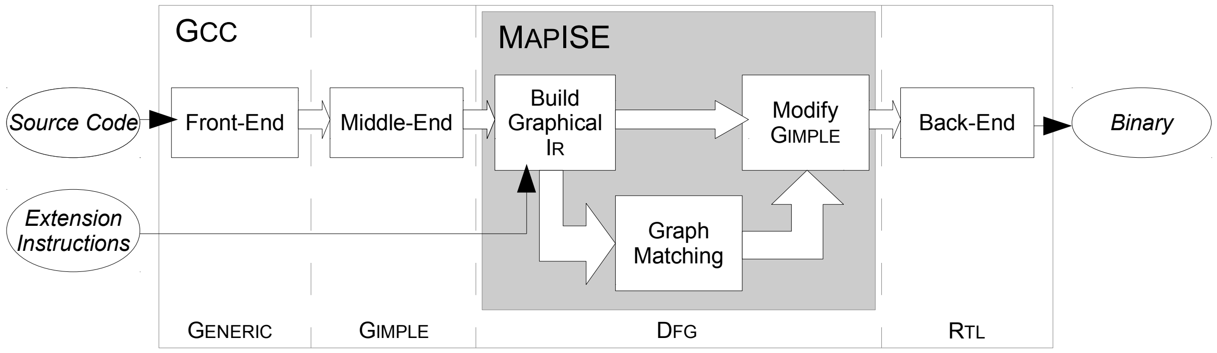


Figure 3: The structure of passes within GCC and MAPISE.

be implemented.

Every node in the basic block implements some simple operation, in the model there are hardware and software costs for each operation. The hardware cost is the fraction of a cycle (maybe greater than one cycle) that it takes to perform the operation. The software cost is the number of cycles the standard instruction that performs this operation takes to complete. Essentially, the AISE algorithm generates data-flow graph templates by iteratively attempting to grow extension instructions. The benefit of adding each node to the current "cut" (instruction) is considered by looking for instructions which will replace many software cycles with a few hardware cycles. Following the generation of templates from basic blocks, the templates are checked for isomorphism with one another so as duplicates may be eliminated, then ranked using their per-execution gain.

3. METHODOLOGY

The tool that implements the techniques described in the paper, MAPISE, works as a series of sub-passes, as shown in figure 3. First a graphical intermediate representation (IR) form is built, then extension instructions are mapped to that. Finally the original GIMPLE is modified.

3.1 Integration into GCC

MAPISE is implemented in GCC 4.2. GCC operates in four main stages: a front-end, a high-level middle-end, a low-level middle-end and a back-end. The pass presented in this paper, MAPISE, runs at the end of the high-level middle-end while the IR is still in SSA form.

GCC uses several IRs. The front-ends translate the input source code into GENERIC, a high-level IR. This is then lowered into GIMPLE, a medium-level IR used by the high-level middle-end. GIMPLE can be used in both an SSA and a non-SSA form, but most high-level optimization passes operate on the SSA form, including MAPISE. The GIMPLE form is then lowered again into a low-level IR: RTL (register transfer language). Low-level optimization and target specific passes operate on RTL. Finally the back-end performs instruction selection on the RTL form ensuring all operations have a one-to-one mapping with assembly. These operations are also annotated with register and scheduling constraints that are used by the register allocator and the scheduler respectively before the RTL is finally converted into assembly.

The instruction mapper presented in this paper, MAPISE, exploits GCC's support for extended inline assembly. This is required to work around a lack of support for specific features that an extension instruction mapping pass requires. The most unusual requirement is the need to support an arbitrary number of extension units with a single compiler binary. If any changes are made to a GCC back-end then GCC must be recompiled. A processor designer or application developer may be evaluating several hundred extension configurations as a design space exploration exercise. Even if the process is automated this would likely still hinder productivity. The implementation described in this paper avoids this by taking a description of the extension unit as part of its input. This way, the requirement of deploying a single compiler install can be satisfied, while still supporting an arbitrary number of extension units.

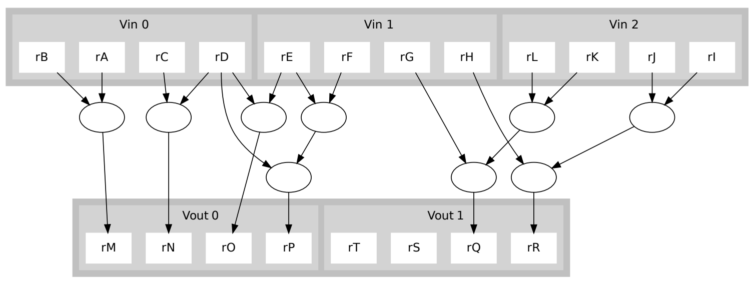
This requirement can be satisfied by inserting ASM statement operations into the IR. These are usually generated by the front-end when a program contains inline assembly. No other passes in GCC insert ASM operands into the IR but the alternative approaches would require GCC to be recompiled with each change. It is worth noting that GCC's extended ASM operations require a precise definition of the operation's dependencies and outputs, this allows the optimization passes to continue to be effective even when ASM statements are present.

If MAPISE mapped any extension instructions onto a function, then once the mapping pass is complete the pass manager is configured to re-run *loop-invariant code-motion* as the mapping pass introduces various temporaries that may benefit from being hoisted.

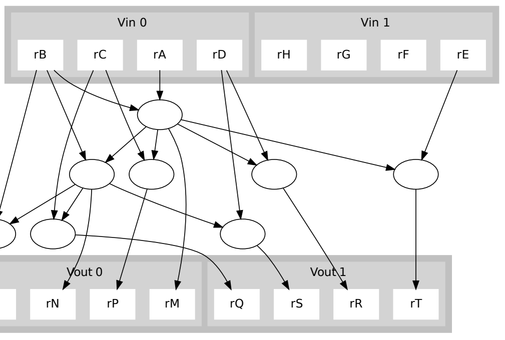
3.2 Construction of Graphical IR

As MAPISE is based on graph-subgraph isomorphism checking, it must operate on a graphical intermediate representation (IR). For this purpose an IR called DFG (Data-Flow Graph) was specified by the author of ISEGEN. As this IR was not going to be used for compilation it did not need to be complete. For a given program the DFG representation is a list of basic blocks with no control-flow information describing how they link together. This is ideal for the purposes of AISE tools since they only look for dataflow graphs within basic blocks, control flow information is extraneous.

Each basic block is a list of nodes and edges and each possible node-type has a one-to-one mapping to some type of GIMPLE node. The converse is not true, however, GIMPLE has many types of nodes not supported by DFG.



(a) A disjoint extension instruction.



(b) An extension instruction where the value saved in `rN` is also used internally.

Figure 4: Example extension instructions automatically generated for an AES benchmark.

For the purposes of AISE, a pass was created in GCC that iterates over the GIMPLE representation of a program and produces a DFG representation. The representation is then serialized into XML and the resultant information transferred to the AISE tools.

MAPISE also needs to operate in DFG because the extension instruction definition is provided to it in XML DFG format. MAPISE therefore processes GIMPLE to create a list of DFG basic blocks and parses the provided XML to create a list of DFG instructions. The manner in which these two tasks occur is completely mechanical and therefore not discussed here. At this stage, however, MAPISE has to do some additional post-processing of the DFG.

Firstly, cast nodes need to be removed as the XML DFG provided will have already had this done on the basis that the hardware implementation of the extension instructions operate on 32-bit arithmetic. Secondly, pointer aliasing information is used to add virtual dependencies between DFG nodes if GCC's virtual use and defines state that two nodes are dependent. This means that if operation X writes to memory, and operation Y reads from what may be the same memory location then Y has a virtual dependence on X. This means that Y must occur after X.

Finally, the VFLIB implementation of the VF2 algorithm [8] that is used to perform graph-subgraph isomorphism checking requires graphs to be in its own format, so for every basic block and every extension instruction definition an additional representation is built.

3.3 Matching Subgraphs

To find where extension instructions may be mapped, a greedy search strategy is used. The extension instructions are sorted by their expected gains. Then, for each basic block every instruction is iterated over. The basic blocks are simply iterated over from start to end, the instructions, however, are ordered according to their expected benefit – the best instructions are considered first. Each basic block and extension instruction pair is passed to VFLIB and if the instruction is a sub-graph of the basic block then a match is recorded. Every DFG node that was just mapped to an extension instruction gets marked as such to avoid a node being mapped to multiple instructions. The same pattern that was just mapped is retried, in case the same pattern may be reused. This process repeats until every extension instruction has been checked.

When VFLIB finds a place to map an extension instruction, it is necessary to check the mapping is viable prior to its application. As a consequence of MAPISE supporting disjoint extension instructions it is possible for VFLIB to find mappings which violate convexity constraints. Convexity violations can occur when one of two (or more) disjoint parts of an instruction become indirectly dependent on the other part of the instruction. The result is that the extension instruction must be scheduled both before and after the intermediary node(s): a clear impossibility. This can only happen with extension instructions which contain disjoint subgraphs, such as in the instruction in figure 4(a). It contains three unconnected sub-graphs and according to the graph-subgraph isomorphism definition (and thus also the VF2 algorithm) each subgraph may be mapped to anywhere in the graph, irrespective of indirect dependencies which means that convexity violations are possible. If convexity constraints are violated then the key offending node is marked as unusable and the extension instruction is tried again. This is allowed to occur a maximum of 10,000 times per basic block/extension instruction pair. Convexity violations are quite common: across the set of 179 benchmarks it was observed that mapping was re-run 279,851 times due to convexity violations and the 10,000 iteration limit was reached 97 times.

3.4 Determining if Two Nodes are Equivalent

A function is provided to VFLIB that when given one node from the graph (basic block) and one node from the subgraph (extension instruction definition) it determines whether or not they are equivalent. As this function is performance critical (it accounts for 46% of MAPISE'S run-time, see table 1) it attempts to establish non-equivalence as quickly as possible. This does not change the complexity of the problem as, in algorithmic terms, the node equivalence function can be thought of as having a constant execution time. In practice, however, reducing the average duration of this constant-time significantly reduces the run-time of MAPISE as this function is called so frequently.

Various other properties must be satisfied for two nodes to be considered equivalent:

- If the nodes do not perform the same operation, have different data types or the basic block node takes a 64-bit value as input, then the nodes are not equivalent. If these nodes are constants then they must have the same value, or they are not equivalent.

- If multiple extension instruction nodes read from a single input register, such as `rD` or `rE` in figure 4(a), then an additional node must be inserted into the extension instruction definition to merge the two edges. Without this the 12-input instruction in figure 4(a) may attempt to fit up to 15 values into 12 slots.
- If the basic block node has already been mapped to another extension instruction then it cannot be mapped to this one. If the SSA name that the basic block node writes to is used outside of the current basic block then the extension instruction node must write its value to an output or they are not equivalent. For example, in the instruction in figure 4(b) there is a node which is connected to both another internal node and the output register `rN`. If a result of a basic block node is used in a different basic block then that node could be mapped to that instruction node, but no other internal node. If it was mapped to different internal node then the value would never be saved and would thus not be accessible when later required.
- The two nodes must have the same number of inputs and the same number of outputs. If a node has two inputs then the predecessors of both nodes must be similar (same operator type, same data type). This is necessary because the VF2 algorithm does not consider edge order, it would consider $A-B$ to be equivalent to $B-A$. If the current node is commutative and the predecessors do not match then they may be swapped and rechecked.

3.5 Exploiting Matches

DFG is an incomplete IR and therefore it is not possible to convert it back into GIMPLE. Since every DFG node is linked with a GIMPLE node, the mappings annotated on the DFG can be used to determine which parts of the GIMPLE to remove instead.

A side-effect of performing mapping on a graphical form, and then inserting the extension instruction into the linear IR is that after the inline ASM has been inserted the IR may no longer be in a valid schedule. It is, however, guaranteed that a schedule exists, therefore a simple scheduling pass is used to reorder the GIMPLE into a valid schedule. This is not associated with the instruction scheduling performed by the back-end since there is no concept of latency in GIMPLE.

4. EMPIRICAL EVALUATION

4.1 Experimental Setup

To obtain performance results a hardware verified cycle-accurate simulator [6] for the ENCORE extensible processor [1] is used. The ENCORE is largely compatible with the SYNOPSYS ARC 700 processor. Each of the 179 benchmarks had cycle counter annotations added to them so as I/O or book-keeping code could be excluded from the performance evaluation. Each benchmark was compiled with vanilla GCC and with MAPISE. The speed-ups presented are the performance of the MAPISE produced code relative to the GCC code. The simulator has been extended with a vector register based encoding which is used to allow a 32-bit instruction to have up to 12-inputs and 8-outputs. Note that this means that the extended register file has 3 read-ports and 2 write-ports, each 128-bits wide.

The compiler options used for compiling each benchmark are `-O2` and `-fprofile-use`. Profiling information is included because ISEGEN also uses this information. Both tools are influenced by GCC's optimization decisions. When profiling data is available GCC will treat hot and cold blocks differently, applying different compiler transforms. If MAPISE is to see the same view of the code that ISEGEN does then it must allow GCC to use profiling data (even if it does not use it itself).

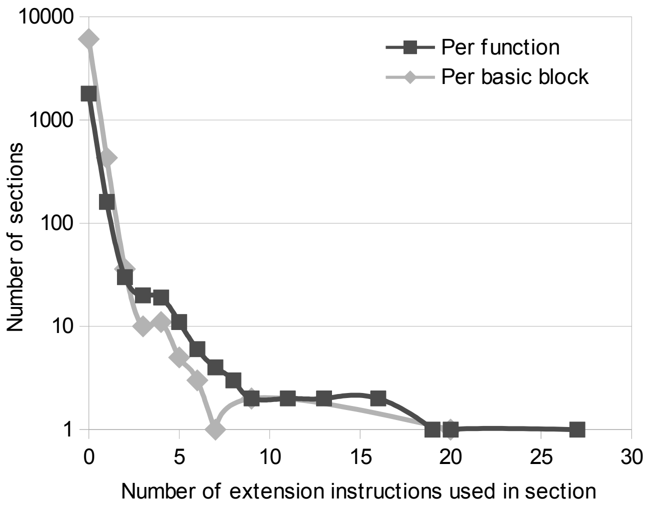


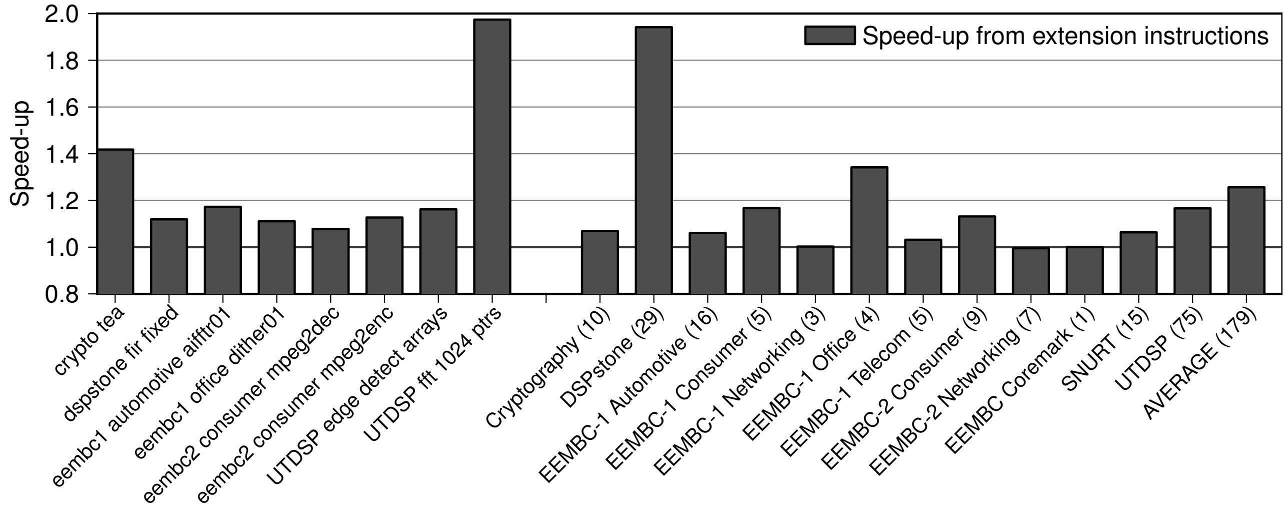
Figure 5: The distribution of the number of extension instructions mapped per-function and per-basic block. The results are from the aggregate of all 179 benchmarks.

The experiments were run on a Linux system with two dual-core 3.0GHz Intel Xeon processors and 4GB of memory. Though only a single core was used to ensure timing consistency.

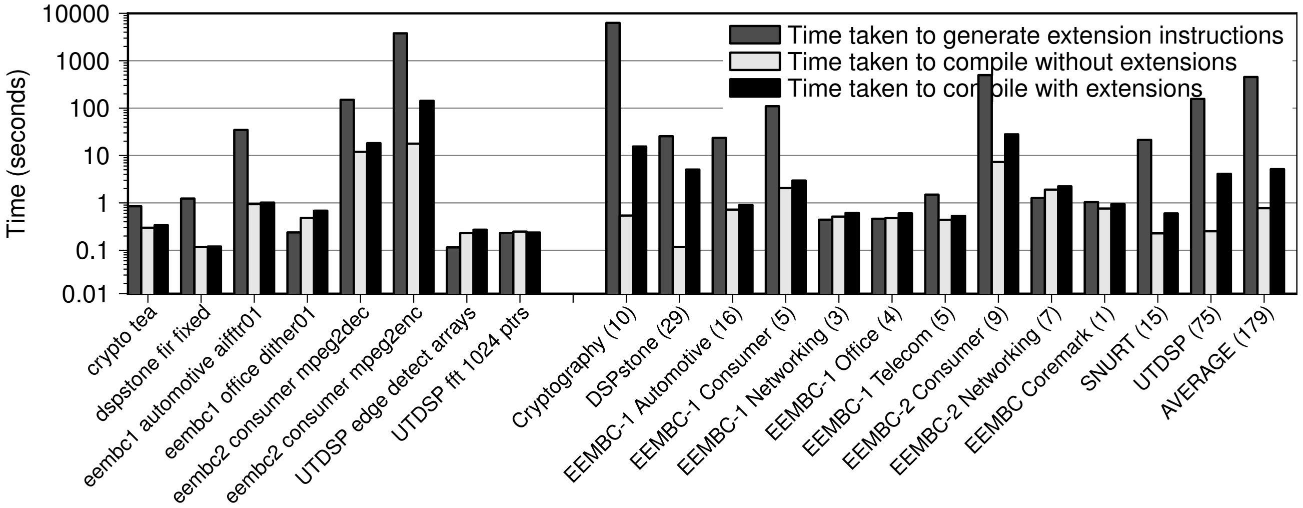
4.2 Summary of Key Results

Figure 6(a) shows the performance speedups achieved when using MAPISE to map extension instructions to the benchmarks that they were generated from. Some representative individual results are presented, as well as aggregate results for each benchmark suite. The average speed-up obtained over the full set of 179 benchmarks is 1.26, though it can be seen that certain suites do not benefit from AISE (e.g. networking benchmarks). Figure 5 shows the distribution of the number of extension instructions mapped to each basic block or function. A large number of basic blocks and function do not use any extension instructions at all, this is expected: a large portion of each benchmark will be concerned with non-critical tasks which will not benefit from extension instructions.

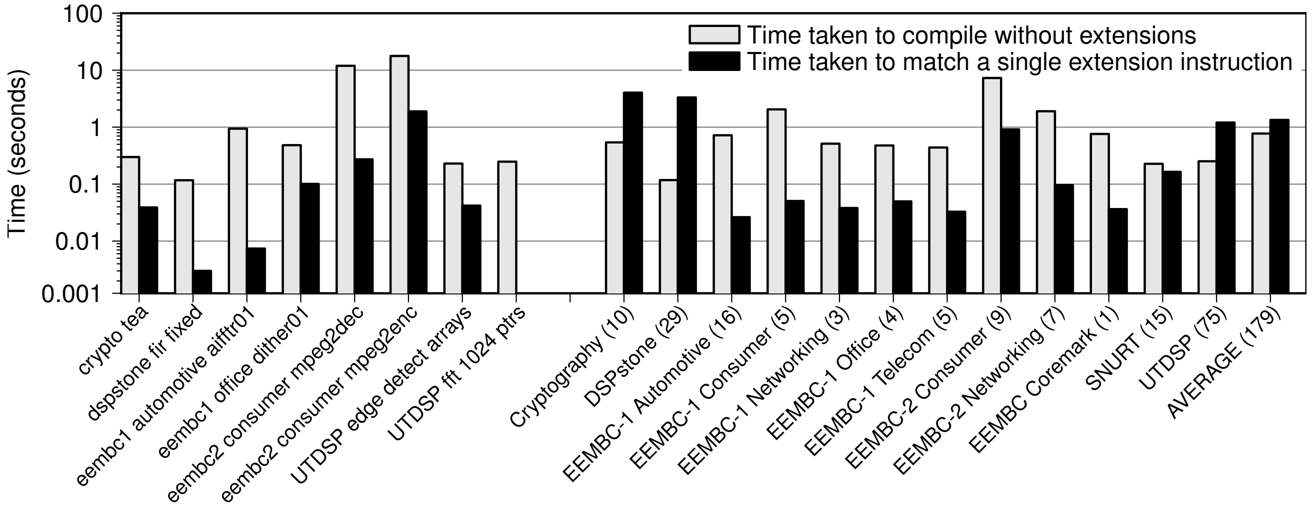
The run-times of various tools are presented in figure 6(b). The left-hand bar for each benchmark or suite is the time it takes for ISEGEN to identify a set of extension instructions for that benchmark. The middle bar is the time that a standard vanilla compile, without extension instructions, takes for each benchmark – whereas the right-hand bar is the length of time the same task takes when mapping extension instructions. It can be seen that although MAPISE is slower than a vanilla compile (an average compile time of 5.2 seconds instead of 0.8) it never has an unacceptable run-time. Even a large benchmark like an MPEG-2 encoder only takes 142 seconds to compile. It is also far quicker than generating instructions, which takes 453 seconds on average, or 3793 seconds for the MPEG-2 encoder. Table 1 breaks down how long MAPISE spends on each of its sub-passes. Clearly the actual instruction mapping



(a) Speed-up due to using extension instructions.



(b) The run-time of various tools.



(c) Time to compile a benchmark without extensions compared to the average time to map one extension instruction to entire benchmark.

Figure 6: The results of using extension instructions. A few representative results are shown on the left of the break, on the right are aggregate results for each benchmark suite. The number in the brackets is the number of benchmarks in that suite.

Task	Time (s)	Time (%)	Sub-Task	Time (s)	Time (%)
Build IRs	27.49	0.9%	Parse XML Instructions	9.33	0.3%
			Build DFG	5.09	0.2%
			Clean BB graphs	7.97	0.3%
			Build VF2 graphs	5.1	0.2%
Mapping	2,904.3	96.1%	VF2 Algorithm	1,482.6	49.1%
			Node Comparison	1,384.9	45.8%
			Viability Checking	33.07	1.1%
Register Allocation	6.92	0.2%			
GIMPLE Modification	83.43	2.8%	Scheduling	82.95	2.7%

Table 1: The total time spent in each sub-pass of MAPISE. The timings are from the summation of all 179 benchmarks.

task dominates, the tool spends 49.1% of its time in the internals of the VF2 algorithm. Additionally that algorithm makes use of the node comparison functions, which account for an additional 45.8% of the run-time. Figure 6(c) normalizes MAPISE'S run-time for a single extension instruction (the MAPISE run-time is divided by the number of extension instructions processed). This demonstrates that on a per-extension-instruction basis small benchmarks (e.g. DSPStone) can take longer for MAPISE to process than large benchmarks (e.g. EEMBC-1 Consumer). For small kernel-based benchmarks GCC will tend to unroll inner loops, creating large basic-blocks which the small number of extension instructions will map to. For large benchmarks, however, most extension instructions will be found to trivially not-map to most of the basic blocks as they are only mappable to a small number of sites. So the complexity of the graph-subgraph isomorphism problem for each basic-block and extension instruction pair strongly influences MAPISE'S run-time.

4.3 Re-use of AISEs

The results of re-using extension instructions on benchmarks that are similar (but not identical) to the ones that they were generated for are shown in figure 7. It can be seen that small changes to the code-shape only result in a small reduction in performance, from an average speed-up of 1.16 to 1.13. Though in a few outlying cases the ability to use the extension instructions may be lost completely, e.g. UTDSP *latrnm 32-64 arrays*. This demonstrates that MAPISE is able to effectively use extension instructions even when the original program is modified.

5. RELATED WORK

An early piece of work in the area of complex instruction selection was undertaken by Leupers and Marwedel [15]. They targeted instructions that may be represented as disjoint data-flow trees, i.e. their operations occur in parallel. For example: instead of targeting deep multiply accumulate instructions, they target wide multiply accumulates where the accumulation occurs via an architecturally visible temporary register and the result of the previous multiply is accumulated. The problem is encoded as the set of register transfer paths possible on the processor, and the set of transfers that each instruction implements. Standard optimal tree covering is performed on these register transfers and then an integer linear program is used to find instructions that may cover multiple register transfers. The type of hardware that this technique targets is less common now and small parallel instructions have mostly been replaced with short tree equivalents, reducing the usefulness of the technique since it was developed. The work was later revisited though [14] to target a parallel form of instructions: sub-word SIMD. Sub-word SIMD tries to pack small operations into standard arithmetic instructions, e.g. packing four 8-bit additions into a 32-bit addition. A standard optimal tree covering technique is extended so that instead of finding a single optimal covering it finds all optimal coverings. An integer linear program is then used to find a set of SIMD instructions that is capable of working within the register constraints that sub-word SIMD introduces. This is an interesting extension but is not equivalent to the instruction mapping requirements of this paper: sub-word SIMD instructions are far simpler than AISE generated extension instructions.

Arnold and Corporaal [4] extended the standard optimal dynamic tree programming algorithm to be able to handle instructions with multiple outputs. In principle, the cost function of the standard dynamic algorithm is modified so that when an output of the multiple-output instruction is used as an input to a node the cost is divided by the number of outputs. This breaks the dynamic tree covering algorithm's ability to find an optimal covering but the effects of that are not evaluated in the paper.

Scharwaechter et al. [18] continue the development of complex instruction mappers by extending the idea of a code-generator generator to be able to handle parallel instructions. The extended generator represents basic blocks and instructions as directed acyclic graphs (DAGs). Several heuristics are used to help reduce a worst-case exponential run-time down to an average-case linear run-time. The set of instructions evaluated were the fusion of only two unconnected simple operations. These are far smaller than the extension instructions considered in this paper and could potentially mean this technique is inappropriate for instructions generated for ASIPs.

Ebner et al. [9] produced a technique for matching graph-based instructions based on a SSA representation. The algorithm finds every possible (overlapping) place to use each instruction. This has a worst-case complexity of $O(\binom{n}{k})$ where n is the number of nodes in a basic block and k in the number of nodes in an instruction, however in practice the complexity is much lower. Each assignment then becomes a variable in a partitioned binary quadratic problem, and is either mapped to an integer linear program for solving optimally or is solved heuristically. The technique was evaluated on the ARM ISA, but again this does not contain any particularly complex instructions. Therefore this technique is not shown to work with complex extension instructions. The worst-case complexity of $O(\binom{n}{k})$ makes it very likely the technique will not scale for large instructions, as when the number of nodes in an instruction (k) grows the complexity explodes.

Finally, Clark et al. [7] describe a promising technique that uses

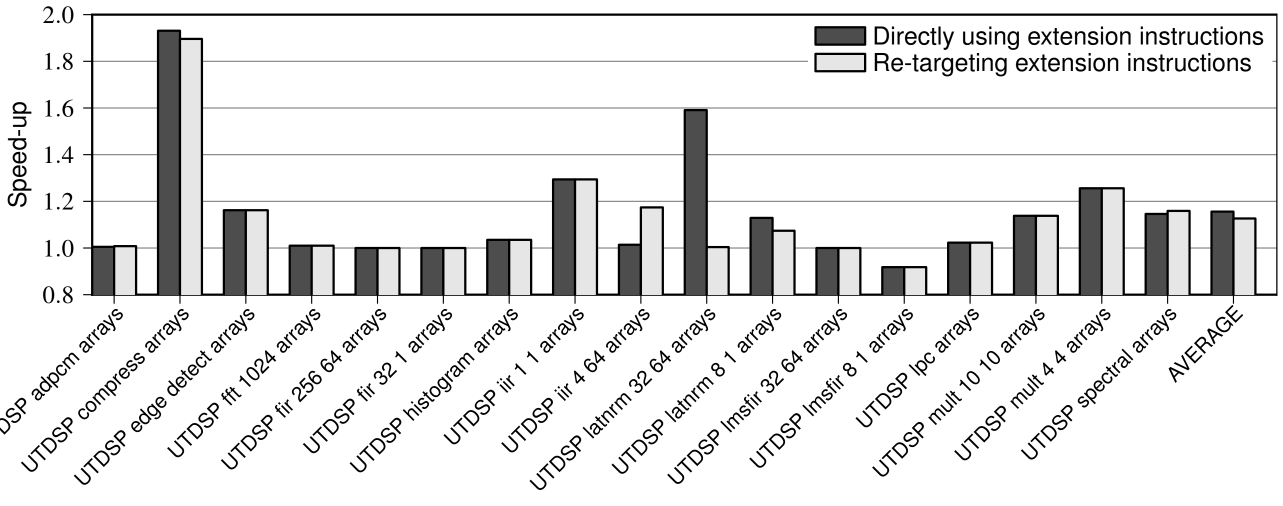


Figure 7: The results of generating extension instructions for UTDSP benchmarks using *pointer arithmetic* but then using them with the *array* version of each benchmark. The left-hand bar for each benchmark shows the speed-up achieved by using instructions generated for each *arrays* benchmark specifically, the right-hand bars show the speed-up achieved by re-targeting the *ptrs* extension instructions.

"Full Enumeration – Unate Covering" to allow effective consideration of multiple extension instructions at once. The run-time requirements are kept under control by heuristically pruning the search space when it gets too large. This is an alternative solution to dealing with the problem of using multiple extension instructions, other than the greedy approach presented in this paper. Unlike this paper, however, it only considers instructions with at most four-inputs and two-outputs (opposed to the twelve-inputs and eight-outputs used in this paper). It also only considers singly connected graphs – the work in this paper allows instructions to be disjoint graphs. It is not clear whether the algorithm's low run times would be maintainable if these features were added.

6. SUMMARY AND CONCLUSIONS

Techniques for automated instruction set extension, which produce large, complex extension instructions, have been shown to generate highly specialized and energy-efficient ASIPs. Without adequate compiler support, however, making use of these instructions is a difficult process. In this paper we have demonstrated that by focusing solely on complex extension instructions a high-level instruction selection pass can use computationally expensive algorithms while maintaining an acceptable run-time. The conventional compiler back-end can then generate scalar code for the remainder of the compilation. We have implemented our novel instruction selection pass in GCC and evaluated it against 179 benchmarks and a total of 1965 automatically generated instructions. Our new code generator is able efficiently exploit complex, automatically generated instruction set extensions and achieves an average speed-up of 1.26 for the ENCORE extensible processor.

Our future work will focus on the integration of the extended compiler into a framework for AISE design space exploration.

References

[1] O. Almer, R. Bennett, I. Böhm, A. Murray, X. Qu, M. Zuluaga, B. Franke, and N. Topham. An end-to-end design flow for automated instruction set extension and complex instruction selection based on GCC. In *Proceedings of the First International Workshop on GCC Research Opportunities (GROW '09)*, pages 49–60, January 2009.

[2] ARC International. ARC FPX white paper, 2007. URL `http://www.arc.com/configurablecores/fpx`.

[3] ARM Ltd. ARM DSP-enhanced extensions, 2001. URL `http://www.arm.com/pdfs/ARM-DSP.pdf`.

[4] M. Arnold and H. Corporaal. Designing domain-specific processors. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES '01)*, pages 61–66, April 2001.

[5] P. Biswas, S. Banerjee, N. D. Dutt, L. Pozzi, and P. Ienne. ISEGEN: an iterative improvement-based ISE generation technique for fast customization of processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 14:754–762, July 2006.

[6] I. Böhm, B. Franke, and N. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC'11)*, 5(4), 2011.

[7] N. Clark, A. Hormati, S. Mahlke, and S. Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *Proceedings of the ACM 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, pages 147–157, October 2006.

[8] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Graph matching: A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[9] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, and A. Kadlec. Generalized instruction selection using SSA-graphs. In *Proceedings of the ACM SIGPLAN/SIGBED Con-*

ference on Languages, Compilers, and Tools for Embedded Systems (LCTES '08), pages 31–40, March 2008.

[10] M. A. Ertl. Optimal code selection in DAGs. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 242–249, New York, NY, USA, 1999. ACM.

[11] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. In *Proceedings of the 4th international workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ARC '08, pages 209–220, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] P. Ienne and R. Leupers. *Customizable Embedded Processors*. Elsevier Inc., 2007.

[13] K. Keutzer, S. Malik, and A. R. Newton. From ASIC to ASIP: The next design discontinuity. In *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, ICCD '02, pages 84–, Washington, DC, USA, 2002. IEEE Computer Society.

[14] R. Leupers and S. Bashford. Graph-based code selection techniques for embedded processors. *ACM Transactions on Design Automation of Electronic Systems*, 5(4):794–814, October 2000.

[15] R. Leupers and P. Marwedel. Instruction selection for embedded DSPs with complex instructions. In *Proceedings of the Conference on European Design Automation (EURO-DAC '96)*, pages 200–205, 1996.

[16] C. Liem, T. C. May, and P. G. Paulin. Instruction-set matching and selection for DSP and ASIP code generation. In R. Werner, editor, *EDAC-ETC-EUROASIC*, pages 31–37. IEEE Computer Society, 1994.

[17] MIPS Technologies. MIPS32(R) architecture for programmers, 2007. URL `http://www.mips.com/`.

[18] H. Scharwaechter, R. Leupers, G. Ascheid, H. Meyr, J. M. Youn, and Y. Paek. A code-generator generator for multi-output instructions. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, pages 131–136, October 2007.

[19] J. M. Youn, J. Lee, Y. Paek, J. Lee, H. Scharwaechter, and R. Leupers. Fast graph-based instruction selection for multi-output instructions. *Softw. Pract. Exper.*, 41:717–736, May 2011.