Instruction Set Extension Identification Under Local, Global, and Solver Execution Time Constraints

Daniel Shapiro	Saurabh Ratti	Miodrag Bolic
dshap092-at-uottawa-dot-ca	…	mbolic-at-site-dot-uottawa-dot-ca

Abstract

The instruction set extension identification problem is the search for a set of custom instructions which can be added to a base processor. A sub-problem is to complete the ISE identification without searching for too long. One aspect of this area of research that needs improvement is the execution time of the sate of the art algorithms which are used to solve this problem. Thus far solutions to large problems are exact but return results too slowly. We propose a non-exact solution in the form of an integer linear program, and proceed to describe our toolchain. We show that heuristics can improve the execution time of the solver by finding an incumbent for each iteration of the ILP. We show the need for a mix of small instructions which are repeated often, and large instructions which although rare provide a large speedup. We propose a method for reducing the execution time of the solver by allowing the user to define a cutoff execution time between improvements in the solution, and a maximum execution time for the solver. Furthermore we define an objective function as the number of cycles saved due to the selection of custom instructions, because this measurement is additive. When identifying ISEs explain how merging identical ones takes advantage of common patterns in the code, and we show how to sum the speedup values now that they are additive. We include in the ISE identification problem some local and global constraints on code defined by the user.

Introduction

Consumer demand is prompting industry to look for ways to increase the execution speed of software applications. One approach to speeding up applications is to partition it into tasks which are scheduled to execute in parallel on separate processors. That approach is called multiprocessing. A further improvement can be achieved by customizing each processor to the code that it will execute. Because it is expensive to verify a processor, designs can be composed of an already verified processor and some custom logic added onto the datapath as custom instructions. A verified processor that can be extended with custom instructions is called an extensible processor. These processors allow a custom instruction datapath to be added around the register bank. Each custom instruction added to an extensible processor is called an Instruction Set Extension (ISE), because it extends the instruction set of the processor [1].

The ISE identification problem is the search for a set of ISEs to implement according to some defined criteria. This is the problem for which we will proceed to propose a solution.

Prior Art

A configurable processor tool-chain converts one or several high-level languages into assembly language instructions and a hardware description of the selected custom instructions (ISEs). The goal of this section is to shed light on some of the current research into configurable processor compilers. This review of prior art is neither complete nor conclusive. A recent review of compilers used for custom embedded systems is available in [33]. Another slightly older reference on the topic is [34]. Some of the common tools and methods employed in this field will be covered.

Several tools are commonly used to generate the hardware description and assembly code for configurable processors. Some of the available compilers to use for a tool-chain are GCC, Trimaran, LCC, CoSy, XPRESS, and Processor Designer [20],[22],[24],[26],[27],[28]. Selecting a compiler to extend is an important aspect of building an ISE identification tool-chain. Some compilers such as XPRESS and Processor Designer are closed source, and therefore can be extended but not modified. Xtensa is the tool that uses XPRESS and it can build a custom instruction tool-chain by profiling user software, or by reading an ISE description provided by the user in the TIE language. The rough equivalent in Processor Designer for ISE description is called LISA 2.0. Two examples of extensions to these tools are [4] and [19]. Open source compilers can be modified and extended at a lower level than the closed source tools and allow researchers to experiment with more advanced ISE identification.

Even if a compiler is open source, it may not be free to use commercially. For example, Trimaran is free to use as a research compiler or for internal corporate use, but cannot be used in a commercially distributed compiler or tool. Another important aspect of compiler selection is the ease of modification and extension. Until the release of GCC 4, GCC was difficult to use even for standard targets available through the compiler. With the release of GCC 4, GCC moved towards a more coherent and modifiable code base [21]. The upgrade to version 4.x included a common intermediate representation for all input languages, and static single assignment. Modern retargetable compilers including GCC 4 and LCC are separated into a front-end that parses the syntax and semantics of software into an intermediate representation (IR), and a backend which consumes IR and emits instructions. Some research has been done into code rewriting at the front-end to improve the results of the ISE identification [8],[19]. Also, research has been done into generating VHDL directly from the control and dataflow graph information [25].

ISE identification and hardware description of the ISEs need not be part of the compiler. Instead, the ISE identification can operate on the assembly code output from the compiler. One option is to match the compiler with a cycle accurate language like Handel-C for hardware generation [12]. In contrast to the idea of using a compiler as a black box, the advantage of using the inside of a compiler to identify custom instructions is twofold. Firstly, high-level optimizations can be integrated into the ISE identification before the architecture has been specified [19]. Secondly, the ISE identification will not be platform specific, and could then be ported to any valid machine description as a library.

The popularity of a compiler is also an important aspect of compiler selection. Your work will be more applicable if it is based on a compiler that is widely available and familiar to your target audience. Furthermore, if you plan to edit the compiler's code base then the size of the code base or the compiler, the size and experience of the development community, and the comprehensiveness of the documentation should all play a role in your decision to choose a compiler. The GCC community is large active and experienced and provides very extensive documentation. It is a popular compiler and can be modified, extended, and redistributed free of charge under the GPL general public license. However, it comes with it a very large code base of several million lines of code. GCC has the added benefit of supporting multiple high-level languages such as C, C++, Java, Ada, FORTRAN, and others, and it supports many target architectures from PowerPC to i386 [21].

One more factor that affects compiler selection is the optimization provided by the compiler. LCC is an example of a non-optimizing compiler, and GCC is an example of an optimizing compiler. Optimization is beneficial because it can reduce code size and execution time. One down side to optimization is that it can reduce the reliability of the line numbers associated with a code segment [31], and we may need those line numbers to exclude some code segments from being implemented as part of an ISE. In GCC 4, Static Single Assignment (SSA) optimization is more consistent when compared to the optimizations provided in GCC 3 because it takes advantage of the common format of all input languages, and it cleanly identifies optimizations such as constant propagation, partial redundancy, dead code, and copy

propagation in the control flow and data flow [30]. In GCC 4 the optimizer is triggered by including '-O' as a command line argument. Figure 1 shows the compilation flow of GCC 4 including the optimization.

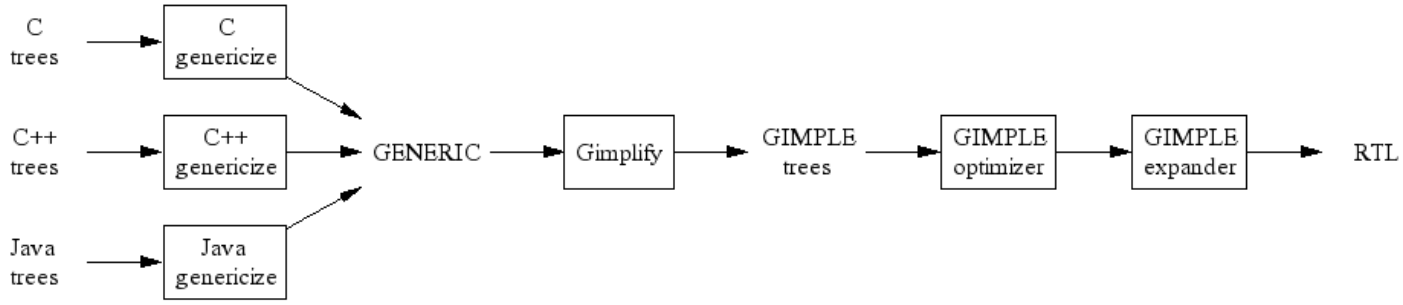


Figure 1: Conversion of GCC frontend trees into a common format (GIMPLE) followed by SSA optimization and register targeting. Taken from [30].

We continue with some of the concepts behind selecting ISEs. When writing a program to identify instruction set extensions, there are several considerations including:

1. Objective when searching for ISEs
2. Granularity of the search space
3. Desired size of ISEs
4. Algorithm type
5. Description of the constraints on the solution
6. Target architecture

The objective is a measure of the fitness of a candidate ISE which enables the comparison of ISEs and the selection of a set from within the ISE search space. The objective may differ across implementations. For example, some solutions take into account the probability that a basic block will be called [3] while others do not [1],[2].

Several authors restrict the ISE search space to the level of basic blocks. A basic block is a list of commands with no jumps or branches in control flow. By looking through the set of the largest Data Flow Graphs (DFG) in a program, good opportunity is provided for finding large ISEs, and the basic block data structure is easily accessible inside most compilers. A DFG is also a Directed Acyclic Graph, or DAG in compiler speak. Programs are composed of DFGs that are connected by control flow. In GCC 4.x a program is represented by a CDFG, which includes control flow and data flow information. Each data flow segment of the CDFG is called a basic block and has only one entry point and one exit point. One way to think of a basic block is like a list of steps with interdependencies. If one instruction in a basic block must be executed, then they must all execute. Some other advantages of exploring only within basic blocks are the ability to ignore control flow and the related complexity in the code, and the availability of a wealth of tools from debuggers to profilers which can analyze basic blocks. Some solutions look beyond the basic block level. For example, in [8] transformations to the Control-Data Flow Graph (CDFG) are used to fuse basic blocks if that fusion will improve the maximum objective function value.

The desired size of an ISE depends upon the application. The authors of [19] assumed that their tool targets application specific programs which will not need to be flexible to future changes. This means that they could search for large ISEs. On the other hand Tensilica assumes that small ISEs are best

because they preserve the generality of a processor to some extent, and a change in the software will not invalidate large parts of the hardware, especially in an ASIC design [19]. These size preferences represent differing philosophies for ISE identification which depend on the application and the desires of the company using the tool. In [8] transformations are used to increase the size of an ISE if advantageous. Pipelining, must be considered for ISEs with deep levels of operations. Unless the ISE is pipelined, the longest register to register path between functional units will lower the safest available clock rate [15].

Algorithms used to identify ISEs are varied. Various solutions are implemented as recursive search programs [1], iterative search programs [6], and Integer Linear Programs (ILPs) [2],[3]. The advantage of defining the ISE identification as an ILP is the availability of advanced commercial solvers which process well defined problem descriptions. As described in [5] ISE identification algorithms can find ISEs by clustering dependencies into an ISE or by clustering parallel operations. Several authors have described the same constraints on the ISE identification problem when clustering dependencies. The ISE has a limited number of inputs and outputs connected to the processor, and the ISE must be convex on the DFG [1],[2],[3],[6],[7],[9],[13],[15],[16],[19],[23]. Some authors include additional constraints such as [7] which forbids the inclusion of specified nodes in the ISE. In [11] the convexity constraint is not included, but an area constraint on the ISE is included. In [11] a library of common valid patterns is used to improve the rate of ISE identification.

The target architecture of a tool will affect its performance because although a higher speedup may be achieved for a given architecture, it may require a larger footprint on the chip, and therefore fewer cores will fit in a System on Chip (SoC). The number of registers visible to the custom instruction at its input and output is directly related to the ISE identification problem. The increasing the number of registers that can be manipulated in parallel increases the chances of finding a highly parallel ISE. Also, in [5] the trade-off between code size and instruction width is discussed. Some example implementations with VLIW targets are [5] and [17], and implementations with RISC targets are [14] and [19]. In [13] a reconfigurable target is specified. An example of a complete and-to-end system targeting a processor on a Xilinx FPGA can be found in [32].

Data Flow Graph Generator

ISEs are mapped to nodes in a DFG, and each ISE must be convex on the graph. To generate a dataflow graph GCC 4.1.3 was used. SSA format debugging output of a C program was parsed into dataflow, and profiling statistics, while the control flow was captured using another debugging parameter. The debugging output was rendered into images. There were two ways to generate dataflow graphs. We could either parse the assembly output, which is machine specific, or we could parse the SSA form of the program which allows us to build instructions that will work for any machine. However, when following the latter approach the number of input and output ports on the target machine must be specified at this point so that valid ISEs will be identified. We chose to implement a data flow analysis of each basic block similar to the method followed in [29], but based on the SSA form. The code for the DFG analysis is not yet complete.

Constraints

The ISE identification problem is subject to several constraints. For instance, an ISE must be associated with an unused opcode, it cannot have more inputs than the processor's register bank has outputs, and it cannot have more outputs than the register bank has inputs. As well, the size of all of the ISEs cannot exceed the amount of free chip area (or blocks in an FPGA). Real-time constraints on the execution time of the code segments might also be specified, and the design must meet the real-time constraints.

The limit on the number of parallel register inputs to the ISE is called the input port constraint [1]. This constraint has the following consequences. For example, a constant input is not counted as an input for the input port constraint. This is because it can be implemented as a constant register inside the ISE. We could have explored the relative trade-off of adding the constant to the ISE or increasing the number of inputs, but we leave that for future work, and assume that by always including the constants we will identify larger ISEs by taking advantage of additional inputs. The cost of all child nodes of the constant is increased by the cost of the constant's size in bits. In future work, this methodology should be improved or justified. Also, we could load inputs each cycle, improving the throughput of the system. In [11] merging ISEs is proposed, which saves expensive hardware resources from being duplicated. Some constraints we did not address in our solution include the noise in the system, and the power consumption.

In [7] a method is shown for excluding specific nodes from being added to an ISE. As the problem is solved, the solver must respect the dynamic nature of software. Specifically, all statements in the software must be marked as one of the following before the search for ISEs can begin:

1. The statement will not change and may be implemented partially or fully as hardware.
2. The statement will not change and must be implemented as hardware.
3. The statement may change and must not be implemented as hardware.

We further reasoned that the user cannot wait for too long for a solution to be presented. Therefore, we allow the user the option to specify a time limit on the execution of the solver. If the solver exceeds the limit (perhaps 15 seconds or 1 hour), then the solver must exit and return the best result that was identified thus far. We allow another time limit on the time between improvements in the solution.

Another important point is that speedup is relative. We contend that because different code segments have different real-time constraints, a large speedup in a piece of code far from its real-time constraint is not as important as a small speedup for a code segment that is beyond its real-time constraint. To solve this problem we include any real-time constraints on a code segment in our problem definition. The constraints are used as the minimum acceptable solution in the ISE identification problem. The algorithm used to solve the problem was defined as an Integer Linear Program (ILP) and a PERL wrapper which will both be explained in more detail below.

We needed a way to prevent some nodes from being implemented as hardware. We do this rather crudely by setting the value of the decision variable governing the inclusion or exclusion for those nodes to 0. This forces from our ISE any nodes that must be implemented in software. As well, we sometimes limit the number of nodes allowed in an ISE for reasons explained in the section on improving the solution quality.

Objective Function

We must choose a criteria for ISE selection. In [1] the objective was to update the best solution until the whole solution was either pruned or searched. The objective in [2] was to maximize the reduction in the critical path delay of the ISE. [3] goes a bit further and maximizes the reduction in the critical path delay including the cost of transferring the data to and from the ISE. In [4] the objective was to generate maximal points along the speedup versus hardware cost tradeoff curve. The tool allows the user to look at the generated curve and select a point at which the hardware cost and speedup are

acceptable. The objective in [5] was to "maximize the reduction in code-size without affecting the performance and to minimize the number of new complex instructions generated." In [6] the estimated merit M(C) of cutting an ISE from the datapath is maximized. In [7] a very interesting memory aware objective is used. The access to memory is allowed, and in effect the ISE can include calls to state variables. The implementation excluded access to loops and arrays, and other complex data structures better left to the processor itself. The objective is to maximize the merit of a cut given the added cost of communication with the memory. These are just a few examples of the many possible objectives.

Our solution is to maximize the speedup of the solution similar to [2]. When calculating the speedup the ILP cannot see repeated subgraphs, and so the objective function value must be an additive quantity that can be augmented as new information about an ISE becomes available. We chose the number of saved clock ticks to complete the ISE as the system speedup. Each time the ILP is called, it processes the entire graph to find the highest objective function value and it's corresponding ISE. At this point, the ILP returns the list of nodes which are mapped to the ISE. The PERL wrapper merges the nodes in the ISE into a single node and assigns the composite node its reduced execution time as the software execution time, and the hardware cost as the sum of the hardware costs of the included nodes. Currently the reduced execution time is overly conservative and is calculated based on the number of saved pipeline calls. Instead parallelism of the ISE should be taken into account further reducing the execution time of the ISE to the critical path delay of the ISE datapath.

For example, let us suppose that a logical OR of 2 registers will take 2 fewer pipeline cycles in a RISC CPU if implemented as an ISE. At compile time, we go through the constraints file and if there is a scaling factor of 2 defined for the line of code containing the OR, then the speedup is 2*2 = 4. We felt that dividing time-old/time-new for each ISE costs a lot of CPU time to compute, removes the additivity of speedup calculations, and hides the scalar increase in execution efficiency. We can multiply a node by a scaling factor to imply the added value of speeding up certain segments of a program. In the future we wish to experiment with the frequency of execution of a code segment as a coefficient in the objective function as in [3].

How to Reduce Solver Execution Time

Current solutions to the ISE identification problem have trouble finding solutions for problems with a large number of nodes. For large DAGs we propose that the wrapper which calls the ILP should select an incumbent solution before running the ILP in a solver. Having an incumbent improves the solver execution time by pruning the search space in directions who's maximum speedup cannot exceed that of the incumbent. The incumbent is selected by forcing a random node in the basic block into an ISE and then including all children or all parents of the included nodes until a constraint is about to be violated. We know that it won't be the convexity constraint because successively including all children or all parents of the outer nodes starting at the root forces convexity. And so we check to see when an input or output port constraint or the area constraint is violated. The incumbent solution helps the solver reduce the time required to find the optimum. As an aside, when real-time constraints are defined, we use the real-time constraint as the incumbent unless the incumbent meets the real-time constraint. We only generate an incumbent for DAGs with more than 100 nodes as a rule of thumb, because small graphs will be processed effectively and there is not need to add overhead to the solution time.

The three sources that led to the idea for an incumbent in the ILP are [18], [17] and [11]. Because we noticed that the convexity constraint is the most difficult to satisfy, our incumbent identification

heuristics are all based on proving ahead of time the convexity of the ISE instead of calculating it. Let us examine the two possible incumbent generators in more detail. As explained above, to quickly generate a feasible ISE we must force a random node from the basic block into the ISE, along with all of its descendants until some cutoff depth. We must stop adding nodes to the ISE when the input or output port constraints are violated, or the size of the ISE exceeds the area constraint. The equivalent ILP that finds ISEs by adding ancestor nodes is calculated by forcing a random node of the basic block into the ISE and including all of its ancestors, and then their ancestors, and so on until a constraint is violated.

Some observations about including an incumbent: The solver may not find a solution better than the incumbent even if there is one because sometimes it gives up too quickly. One possible strategy is to gradually reduce the incumbent value to see if a feasible solution exists, and then to increase it and see if the solver picks up where it left off and finds a better solution than the incumbent. One other strategy is to change the internal parameters in the solver to make it look for a solution for a long time. This approach may be a bad idea because sometimes the solver can run basically indefinitely when searching through a large solution space.

Can we stop when the hardware size of all of the ISEs is at or near the available hardware size? We can if we want to, but this is not the optimal solution. We may find larger speedups further down the line of basic blocks, and so we may miss some high speedup ISEs. We need a way to reduce the execution time of continuing to traverse the solution space. So, we let the user decide how exhaustively to search past a feasibly selection of ISEs. As with genetic algorithms or simulated annealing, the number of changes to the selected ISEs reduces as the fitness of the ISEs increases. So, we set a parameter to describe the time between ISE substitutions in seconds as the cutoff to calling another iteration of the ILP. Of course, if the solver completes the search of the solution space, then the solver will stop.

How to Improve Solution Quality

Solvers have a difficult time with integer and binary variables when compared to linear variables. It is prudent to try to include as few binary or integer variables as possible in a problem description. A binary decision variable (Xi) is created for every node (i) in the data flow graph. The solver must search through the possible values of the vector X={X1,X2,..Xi} to find the optimal objective function value. The search is subject to the input port, output port, exclusion (do not include specific nodes), and convexity constraints. Any solution is represented by a binary string X, and must meet all of the constraints in order to be a feasible solution. To reduce the number of integer typed variables, we require that input variables in the ILP such as instruction speed and hardware cost be linear typed numbers but set to integer values. This approach guarantees an integer result even though there are no integer typed variables. As well, all of the constraint variables are binary typed.

A data flow graph may have several instances of a given subgraph, which improves the value of implementing that subgraph as a custom instruction (an ISE). Implementing a repeated subgraph as an ISE improves the speedup associated with an ISE without the added hardware cost, but it is hard to find repeated patterns in graphs. The way we deal with repeated subgraphs is to notice that when identifying the best ISE, and then the second best, and so on, the identical ISEs will come out of the ILP one after the other. This is because they have the same objective function value. We merge the two identical ISEs into a single ISE, sum their speedup values. This can also be achieved if two ISEs from different basic blocks are identical. This concept comes in part from [3].

By selecting first the ISEs with the largest objection function value, large ISEs are favored over small ones. The small ISEs which are very common may not be realized until the very end of an exhaustive search of the solution space because on their own they have a low objective function value. As well, small subgraphs are more likely to repeat than large ones, and they may cost less to implement in hardware as well due to their size. We therefore propose an alteration to the ILP as follows. During even calls to the solver the ILP is forced to search only for ISEs which have "n" nodes or less, for some value n. During odd calls the the ILP, the solution must have more than n nodes. This causes the strong benefit of small ISEs to be expressed earlier in the search of the solution space. If the solver is interrupted, it is now more likely to have a good feasible result than just a few large ISEs.

The order in which the ILP is passed basic blocks for processing is important. We traverse the basic blocks in order of their hardware implementation priority until the time constraint on the solver is exceeded or all basic blocks have been processed. Assignment of hardware resources to basic blocks is prioritized based upon the order defined by the user's goals. The priority of each basic block is equal to the sum of the priority of the nodes in the basic block divided by the number of nodes in the basic block.

How to Improve Usability

The programmer of the software will know what parts of the design are safe to be expressed as hardware, and which parts may change and should be kept in software. In the design of our system we reasoned that software programmers will not want to define pragmas in their code, especially legacy code, and instead we propose that a user interface component in the software IDE should collect the requirements that the user wants to define into a requirements file. The constraints can be displayed graphically in the IDE underneath the code as text-background colors. The requirements module of our system will specify line number ranges and the associated constraints in a file. Using these constraints we can reduce the solution space for the ISE identification problem by elimination from consideration the code that may change, and giving priority to the implementation of some nodes in the DFG. From an optimization point of view the improvement in solver execution time does not differ if pragmas or a graphical tool is used to express the relative safety of implementing a statement in hardware.

We propose the use of a slider-bar to define the relative importance of hardware vs. software implementation. If the slider bar is left in the center, then selected code can be added to an ISE. If the Slider bar is 100% on the hardware side, then the code will be forced into hardware and the area required by that custom instruction is allocated from the count of free area. If the Slider bar is 100% on the software side, then the code is forced into software, which means that the nodes associated with that code are not allowed to be included in an ISE. The gray areas can be used to skew the relative importance of speedup for a given code segment. Moving the slider-bar towards the software side implies lower priority and towards the hardware side implies higher priority. This data is also used for the hardware generation phase of the project which is beyond the scope of this paper. In brief, the relationship between software and hardware in the DSE is simplified by relating all hardware constraints to all software constraints and solving for a single variable.

Overview of Our Toolchain

Our toolchain collects user requirements and software, and outputs an ISE description. The ISE identification ILP was written in the LINGO 10.0 language and executed in the LINGO solver. A lingo ILP is composed of sets, data, an objective and constraints. We defined the objective as the greatest savings in clock ticks, and our constraints on the ISE are on the number of inputs, number of outputs, amount of hardware resources available to implement the ISE, the legality of including a node, the number of nodes in the ISE, and the convexity of the ISE. The convexity constrain takes the most time to satisfy. Currently the toolchain does not include the Eclipse IDE, and instead the code is passed directly to GCC. The overall picture of our tool can be seen in the following figure.

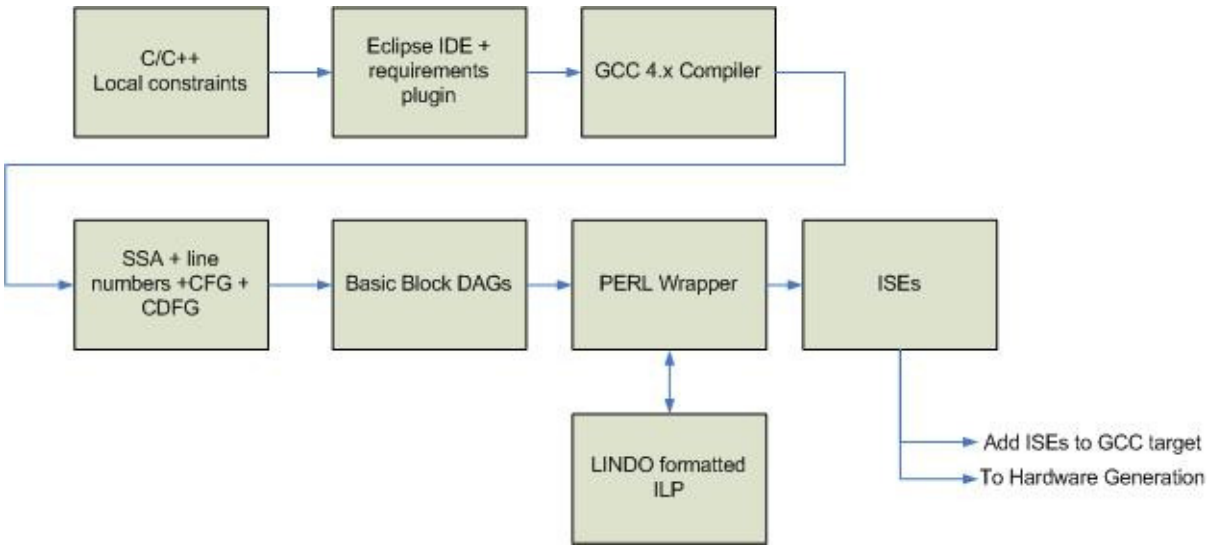


Figure 2: Toolchain overview

The PERL wrapper calls an incumbent generation ILP, followed by an ISE identification ILP which alternately looks for small and large ISEs. When an ISE is identified, the nodes for the ISE are collapsed into a single node. In our ILP there is a set which represents all nodes in the basic block, and another set for the arcs (source, destination) between nodes. A node is associated with an implementation cost and an execution time. The PERL wrapper fills in the available hardware and predefined port constraint. The objective function maximizes the number of saved execution cycles due to the implementation of the ISE. To calculate the saved cycles, we subtract the total time required to execute the nodes in software and subtract the total time required in hardware. In future work, we would like to express the hardware time as the minimum path through the ISE, which is much more realistic.

As described in the section on improving the solution quality, each bit of the binary vector X represents the decision to include or exclude a node in the ISE. Using the algorithm of [3], we find concavity (non-convexity) by identifying nodes not in the ISE that have an ancestor and a descendant that are in the ISE. To force the convexity of an ISE we set as a constraint the sum of all nodes with the concavity condition to 0. The number of ancestors and descendants of a node is forced to a binary value. To calculate the ancestor and descendant relationships we use:

```
@for( NODE( I):A( I)=@if(@sum(ARC(k,I):X(k)+A(k))#ge#1,1,0));
@for( NODE( I):D( I)=@if(@sum(ARC(I,k):X(k)+D(k))#ge#1,1,0));
```

The number of outputs of an ISE is the number of nodes within the ISE having no descendants within the ISE. The number of inputs to an ISE is the number of nodes in the ISE having no parent within the ISE. We set a maximum hardware cost for the ISE, and constants for the maximum number of inputs and outputs. We also set an incumbent for the objective function value which must be exceeded for the solution to be valid (the incumbent speedup).

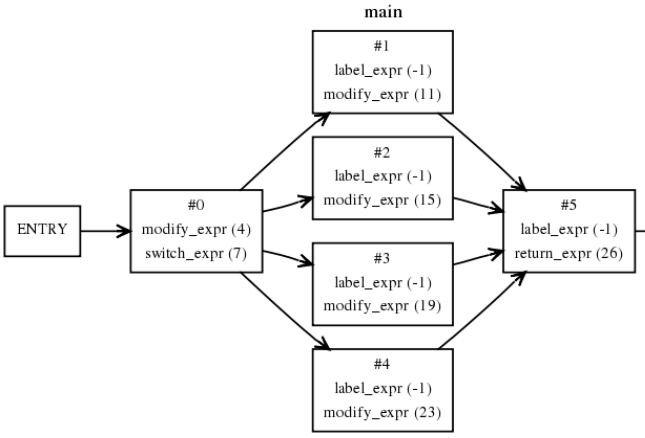
Results

For a simple program with a switch statement, the following output was obtained:

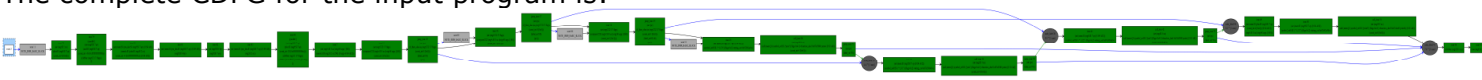
Input C program:

```
main(int argc, char *argv[])
{
        int i,j,k;
        i=1;
        j=1;
        k= (int) argc;
        switch (argc)
        {
                case 1:
                puts("Case 1\n");
                i=j+&k;
                break;
                case 2:
                puts("Case 2\n");
                i=j-k;
                break;
                case 3:
                puts("Case 3\n");
                i=j*k;
                break;
                default:
                printf("The input was %d\n", argc);
                i=2*j+2*k;
                break;
        }
}
```

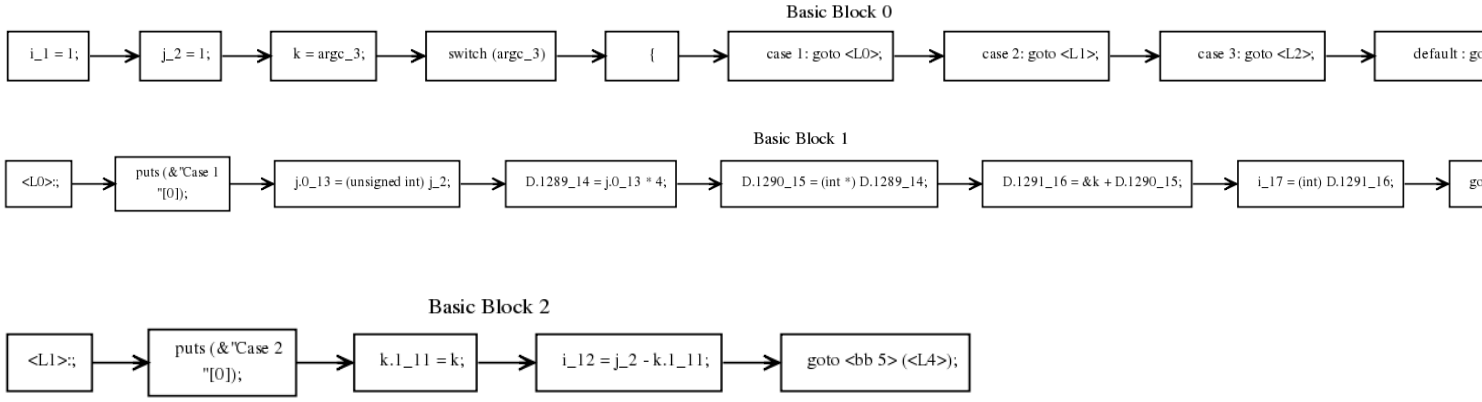
Control flow graph with line numbering of ba blocks:

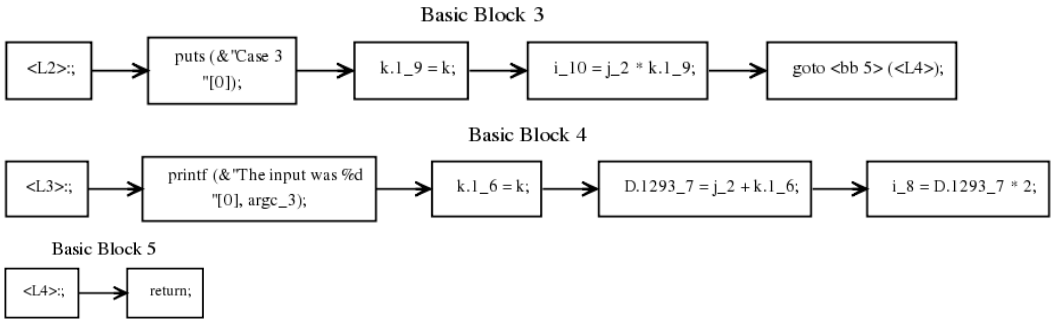


The complete CDFG for the input program is:



Finally, the SSA form statement lists which must now be parsed into dataflow graphs are:





Future work

In the future we may move the LINDO code to AMPL and use the CPLEX solver as [2],[3] have used. As well, we will implement the requirements specification tool in Eclipse IDE, and import GCC into the Eclipse environment as the default compiler. Another task on the horizon is the completion of the conversion of the SSA output from GCC into LINDO or AMPL. There are some interesting ideas about ISE fusion which may be used to reduce the hardware size of ISEs by sharing expensive functional units like multipliers, and we may use this in our toolchain.

Conclusion

A basic non-exact ISE identification tool was implemented based on GCC 4, PERL, and LINDO. Several ideas for improving the solution quality, tool usability, and solver execution time were proposed.

References

[1] Kubilay Atasu, Laura Pozzi, Paolo Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," *International Journal of Parallel Programming,* vol. 31, no. 6, December 2003, pp.441-428.
[2] Kubilay Atasu, Günhan Dündar, Can Özturan, "An integer linear programming approach for identifying instruction-set extensions," in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Jersey City, NJ, September 2005, pp.172-177.
[3] Kubilay Atasu, Robert G. Dimond, Oskar Mencer, Wayne Luk, Can Özturan, Günhan Dündar, "Optimizing instruction-set extensible processors under data bandwidth constraints," in *Proceedings of the conference on Design, automation and test in Europe,* Nice, France, April 2007, pp.588-593.
[4] David Goodwin, Darin Petkov, "Automatic generation of application specific processors," in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems,* San Jose, California, 2007, pp.137-147.

[5] Partha Biswas, Nikil Dutt, "Reducing Code Size for Heterogeneous-Connectivity-Based VLIW DSPs through Synthesis of Instruction Set Extensions," in *CASES'03*, San Jose, California, USA, 2003, October 30–November 2, pp. 104-112.
[6] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Laura Pozzi, Paolo Ienne, "ISEGEN: Generation of High-Quality Instruction Set Extensions by Iterative Improvement," in *DATE'05*, 2005, pp. 1-6.
[7] Partha Biswas, Nikil Dutt, Paolo Ienne, Laura Pozzi, "Automatic Identification of Application-Specific Functional Units with Architecturally Visible Storage," in *DATE06*, 2006, pp. 212-217.
[8] Paolo Bonzini, Laura Pozzi, "Code Transformation Strategies for Extensible Embedded Processors," in *CASES'06*, Seoul, Korea, 2006, October 23–25.
[9] Paolo Bonzini, Laura Pozzi, "Polynomial-Time Subgraph Enumeration for Automated Instruction Set Extension," in *DATE07*, 2007, pp.1331-1336.
[10] Nathan Clark, Hongtao Zhong, Scott Mahlke, "Processor Acceleration Through Automated Instruction Set Customization," in *Proceedings of the 36th International Symposium on*

Microarchitecture, 2003.
[11] Jason Cong, Yiping Fan, Guoling Han, Zhiru Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures," in *FPGA'04*, Monterey, California, USA, 2004, February 22–24, pp. 183-189.

[12] Robert Dimond, *Compilation and Simulation Support for Custom Instruction Processors*, B. Eng., Information Systems Engineering, Imperial College, London, England, 2003.
[13] Carlo Galuzzi, Elena Moscu Panainte, Yana Yankova, Koen Bertels, Stamatis Vassiliadis, "Automatic Selection of Application-Specific Instruction-Set Extensions," in *CODES+ISSS'06*, Seoul, Korea, 2006, October 22–25, pp. 160-165.
[14] Valentina Salapura, Michael Gschwind, "Hardware/Software Co-Design of a Fuzzy RISC Processor," in *DATE98*, 1998.
[15] Wayne Luk, Kubilay Atasu, Rob Dimond, Oskar Mencer, "Towards Optimal Custom Instruction Processors," presented at *HOT CHIPS 18*, Palo Alto, California, USA, 2006, August 21-22.
[16] Laura Pozzi, Paolo Ienne, "Exploiting Pipelining to Relax Register-File Port Constraints of Instruction-Set Extensions," in *CASES'05*, San Francisco, California, USA, 2005, September 24–27, pp. 2-10.

[17] C. Alippi, W. Fornaciari, L. Pozzi, M. Sami, "A DAG based design approach for reconfigurable VLIW processors," in *DATE 1999*, 1999, , March, pp. 778–79.

[18] John W. Chinneck, "Practical Optimization: A Gentle Introduction" [online], Ottawa, Ontario, Canada: Systems and Computer Engineering, Carleton University, Nov. 2007 [cited Nov. 27, 2007], available from World Wide Web: <http://www.sce.carleton.ca/faculty/chinneck/po.html>.

[19] Richard Vincent, Bennett Alastair, Colin Murray, Bjorn Franke, Nigel Topham, "Combining Source-to-Source Transformations and Processor Instruction Set Extensions for the Automated Design-Space Exploration of Embedded Systems," in *LCTES'07*, San Diego, California, USA, 2007, June 13–15, pp. 83-92.

[20] "Using the GNU Compiler Collection" [online], Nov. 2007 [cited Nov. 27, 2007], available from World Wide Web: <http://gcc.gnu.org/onlinedocs/gcc-4.2.2/gcc.pdf>.
[21] Paolo Bonzini, "Using GCC as a research compiler," presented January 13th, 2006.
[22] "An Infrastructure for Research in Backend Compilation and Architecture Exploration" [online], Nov. 2007 [cited Nov. 27, 2007], available from World Wide Web: <http://www.trimaran.org>.

[23] R. Leupers, K. Karuri, S. Kraemer, M. Pandey, "A Design Flow for Configurable Embedded Processors based on Optimized Instruction Set Extension Synthesis," in *DATE06*, 2006, pp. 581-586.

[24] David R. Hanson and Christopher W. Fraser, *A Retargetable C Compiler: Design and Implementation*, 1st edition, Addison-Wesley Professional, January 31, 1995.

[25] Jinhwan Jeon, Yongjin Ahn, Kiyoung Choi, "CDFG Toolkit User's Guide," School of Electronic Engineering, School of Electronic Engineering,
Seoul National University, Seoul, Korea, Report No. SNU-EE-TR-2002-8, Aug. 2002.

[26] "CoSy compiler development system" [online], Nov. 2007 [cited Nov. 27, 2007], available from World Wide Web: <http://www.ace.nl/compiler/cosy.html>.

[27] "Processor Developer's Toolkit Product Brief" [online], Santa Clara, Calif.: Tensilica Inc., 2007 [cited Nov. 27, 2007], available from World Wide Web:
<http://www.tensilica.com/pdf/processor_dev_toolkit.pdf>.

[28] "CoWare® Processor Designer Programmable Accelerators for Platform-Driven ESL Design" [online], San Jose, Calif.: Coware Inc., 2006 [cited Nov. 27, 2007], available from World Wide Web:
<http://www.coware.com/PDF/products/ProcessorDesigner.pdf>.

[29] Uday Khedker, "Implementing a Data Flow Analysis Pass in GCC," presented at *Workshop on GCC*

Internals, 2007, June.

[30] Diego Novillo, "Tree SSA A New Optimization Infrastructure for GCC," in *Proceedings of the 2003 GCC Developers Summit*, Ottawa, Ontario, Canada, 2003, pp. 181-194.

[31] Charles Fischer, "CS 701 Charles N. Fischer Fall 2007" [online]: Fall 2007 [cited Nov. 27, 2007], pp. 15, available from World Wide Web: <http://pages.cs.wisc.edu/~fischer/cs701.html>.

[32] Partha Biswas, Sudarshan Banerjee, and Nikil Dutt, "Processor Customization on a Xilinx Multimedia Board," Center for Embedded Computer Systems School of Information and Computer Science, University of California, Irvine, CA 92697, USA, Report No. 06-04, Mar. 2006.

[33] R. Leupers, M. Hohenauer, J. Ceng, H. Scharwaechter, H. Meyr, G. Ascheid and G. Braun, "Retargetable compilers and architecture exploration for embedded processors," in *IEE Proc.-Comput. Digit. Tech.*, Vol. 152, No. 2, March 2005, pp. 209-223.

[34] R. Leupers, "Compiler Design Issues for Embedded Processors," in *IEEE Design & Test of Computers*, July–August 2002, pp. 2-9.