University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

January 2008

Encoding Mini-Graphs with Handle Prefix Outlining

Anne W. Bracy
University of Pennsylvania

Amir Roth
University of Pennsylvania, amir@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
Bracy, Anne W. and Roth, Amir, "Encoding Mini-Graphs with Handle Prefix Outlining" (2008). *Technical Reports (CIS)*. Paper 891.
http://repository.upenn.edu/cis_reports/891

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-36

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/891
For more information, please contact repository@pobox.upenn.edu.

Encoding Mini-Graphs with Handle Prefix Outlining

Abstract
Recently proposed techniques like mini-graphs, CCA-subgraphs, and static strands exploit application-specific compound or fused instructions to reduce execution time, energy consumption, and/or processor complexity. To achieve their full potential, these techniques rely on static tools to identify common instruction sequences that make good fusion candidates. As a result, they also rely on ISA extension facilities that can encode these chosen instruction groups in a way that supports efficient execution on fusion-enabled hardware as well as compatibility across different implementations, including fusion-agnostic implementations.

This paper describes *handle prefix outlining*, the ISA extension scheme used by mini-graph processors. Handle prefix outlining can be thought of as a hybrid of the encoding scheme used by three previous instruction aggregation techniques: PRISC, static strands, and CCA-subgraphs. It combines the best features of each scheme to deliver both full compatibility and execution efficiency on fusion-enabled processors.

Keywords
Instruction fusion, ISA extension, outlining

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-36

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/891

Encoding Mini-Graphs with Handle Prefix Outlining

Anne Weinberger Bracy and Amir Roth

***Abstract*— Recently proposed techniques like mini-graphs, CCA-subgraphs, and static strands exploit application-specific compound or fused instructions to reduce execution time, energy consumption, and/or processor complexity. To achieve their full potential, these techniques rely on static tools to identify common instruction sequences that make good fusion candidates. As a result, they also rely on ISA extension facilities that can encode these chosen instruction groups in a way that supports efficient execution on fusion-enabled hardware as well as compatibility across different implementations, including fusion-agnostic implementations.**

This paper describes *handle prefix outlining*, the ISA extension scheme used by mini-graph processors. Handle prefix outlining can be thought of as a hybrid of the encoding scheme used by three previous instruction aggregation techniques: PRISC, static strands, and CCA-subgraphs. It combines the best features of each scheme to deliver both full compatibility and execution efficiency on fusion-enabled processors.

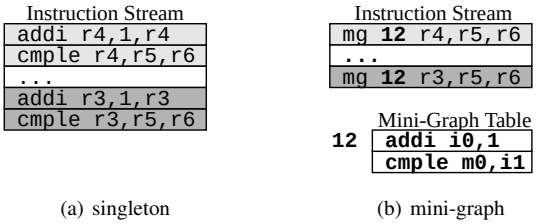
***Index Terms*— Instruction fusion, ISA extension, outlining.**

I. Instruction Fusion and Mini-Graph Processing

MINI-GRAPH processing is a form of instruction fusion (the grouping of multiple operations into a single processing unit) that increases the efficiency of superscalar designs [**?**], [**?**]. Rather than targeting performance improvement by executing fused instructions on custom functional units, mini-graph processing targets performance efficiency via resource amplification. Because a mini-graph processor operates on fused instructions rather than singleton instructions throughout the pipeline, the capacity and bandwidth of structures that manipulate instructions can be reduced with no performance penalty (on average). Mini-graph processing is the logical generalization and extension of micro-op fusion, a technique used in recent Intel processors [**?**]. It extends the benefits of fusion to more structures and pipeline stages and to more dynamic instructions.

Mini-graph processing works by changing the internal representation of fused instruction groups. Inside the pipeline, an N-instruction mini-graph is represented as a single instruction *handle* and an N-instruction *template*. The handle encodes the mini-graph's interface: it includes a special opcode, the mini-graph's interface registers, and an immediate value (MGID) that identifies the mini-graphs template. The template is sequential micro-code that implements the instruction sequence represented by the mini-graph: it encodes the opcodes and immediates of the original instructions and their register dataflow. Multiple static mini-graphs with the same dataflow shape and immediate inputs can share templates. An on-chip memory called the mini-graph table (MGT) caches the most recently used templates.

Figure **??** shows two two-instruction sequences whose dataflow graphs have the same shape, operations (`addi` and `cmple`), and immediate operands, (`1`), but use different register names. Figure **??** shows the same instruction stream executing on a mini-graph processor. Each two-instruction sequence is replaced with a single-instruction handle. Because the original sequences can share a mini-graph template, the handles have the same MGID (`12`). Each handle includes the register names used in the original sequence as parameters. In the MGT, logical register names are replaced by specifiers that denote a parameterized input or output or a value produced by an earlier instruction in the mini-graph.



(a) singleton (b) mini-graph

Fig. 1. Mini-Graph Representation.

The change of representation from instruction sequence to handle-template is the key to amplification. Most of the pipeline processes the handle; the template is invoked only during execution. The handle representation itself amplifies issue queue and reorder buffer capacity as well as decode, issue, and commit bandwidths. A statically enforced atomicity restriction means that mini-graph "interior" register communication can take place without actual registers and enables amplification of register file capacity and read/write bandwidth as well as rename bandwidth. Amplification is proportional to *coverage*, the percent of dynamic instructions that are embedded in mini-graphs. Each of the mini-graphs in Figure **??** provides coverage of one dynamic instruction per instance.

II. Encoding Scheme Criteria

A mini-graph encoding scheme is responsible for encoding the statically-chosen mini-graphs into the binary in a way that enables the split handle-template representation to be constructed at runtime. This section lists the desired attributes of an encoding scheme. The attributes are sorted roughly in descending order of importance *in the context of mini-graph processing*. Mini-graph processing differs from other forms of instruction fusion in that it emphasizes capacity and bandwidth amplification and that it targets general-purpose implementations where compatibility is valued. We define two primary criteria:

- **Instruction cache capacity and fetch bandwidth amplification.** In the context of mini-graphs this is the most important feature. Supporting this feature essentially requires representing each mini-graph as a single instruction in the instruction cache.
- **Support for singleton execution of mini-graphs.** This feature has several uses. It provides functional compatibility across different mini-graph processor implementations. It potentially simplifies the handling of exceptions that occur in mini-graphs. It potentially simplifies debugging on a mini-graph processor. And it provides a mechanism for minimiz-

ing MGT misses. Specifically, mini-graphs that overflow the MGT can simply execute in singleton form.

We also define several secondary criteria:

- **Support for and control over template sharing.** Template sharing is potentially important for making efficient use of the MGT and for minimizing MGT misses.
- **Control over mapping of template to MGT set.** Explicit control over MGID assignment is another feature that is potentially important in increasing MGT efficiency and avoiding MGT misses. A simple policy assigns MGIDs to templates in order of decreasing coverage. Templates with MGIDs that are larger than the number of entries in the MGT execute in singleton form. A more sophisticated policy could assign templates from different program phases to MGIDs that would map to the same MGT set, a scheme analogous to instruction cache conscious code placement [**?**].
- **Functional compatibility with non mini-graph processors.** This is a superset of the ability to execute mini-graphs in singleton form. It implies that mini-graph execution in singleton form should be the default behavior if mini-graph annotations are simply ignored, i.e., treated as nops.
- **Simplicity of dynamically constructing handle-template representation.** If the handle-template representation is not explicit in the binary, then it must be constructed at runtime. The most difficult aspect of constructing this representation is determining the identity of the "liveout" register.

Finally, we define two tertiary criteria or non-criteria:

- **MGT miss penalty.** Control over template sharing and MGT indexing help maximize amplification while reducing MGT misses. However, some MGT misses are unavoidable.
- **Performance of singleton execution of mini-graphs.** Like MGT misses, singleton execution of a mini-graph should also be a rare event. Mini-graphs must execute in singleton form when a mini-graph binary is executed on a non-mini-graph processor. We consider it unlikely that a user will execute a mini-graph enabled binary without having access to *both* a mini-graph processor *and* the original singleton binary, and very unlikely that she will have access to neither. Even on a mini-graph processor, the dynamic instances of a particular mini-graph may need to execute in singleton form if the processor does not support the particular template. Or if the number of templates exceed the capacity of the MGT. We consider these to be unlikely scenarios as well, as we asssume that mini-graph code will be created specifically for the processor on which it is intended to run, perhaps even "just in time". An individual dynamic mini-graph may need to execute in singleton form for debugging purposes or to simplify the handling of a difficult exception that is embedded in it. Again, both of these scenarios are rare.

III. Encoding Schemes

This section describes four mini-graph encoding schemes. The first three were proposed for use with other fusion techniques, but we describe them in the context of mini-graphs here. The fourth is the one used by mini-graphs. Figure **??** shows the mini-graph from Figure **??** as it would be represented by each encoding scheme, both in the binary and in the instruction cache.

A. PRISC: Handle Replacement

Handle replacement is the name we give the PRISC encoding scheme as it would apply to mini-graphs [**?**]. Each mini-graph is replaced with the corresponding handle; the templates are stored in a new section in the binary (e.g., `.mg`). section. On a miss, the MGT loads the appropriate template from this section. This scheme resembles the DISE [**?**]-based scheme described in the initial mini-graph proposal [?]. Figure **??** illustrates.

Handle replacement has several advantages. It obviously supports amplification—each mini-graph is represented in the instruction cache as a single instruction. The fact that handle-template representation is explicit in the binary makes it trivial to construct that representation at runtime. Handle replacement also allows explicit control over template sharing and MGT placement. And while MGT contents can be managed precisely to reduce misses, MGT miss cost is also moderately low. The PRISC proposal suggests using an exception handler to fill the template cache. With a hardware implementation of this handler—similar to the x86 TLB miss handler—the cost of a miss would roughly equal the cost of an L2 access.

The primary drawback of handle replacement is its inability to support singleton execution of mini-graphs. This jeopardizes compatibility across both mini-graph and non-mini-graph processors and potentially complicates exception handling. These were lesser issues for PRISC itself which targeted application-specific processors and limited fusion to integer operations.

B. Static Strands: Prefix Tagging

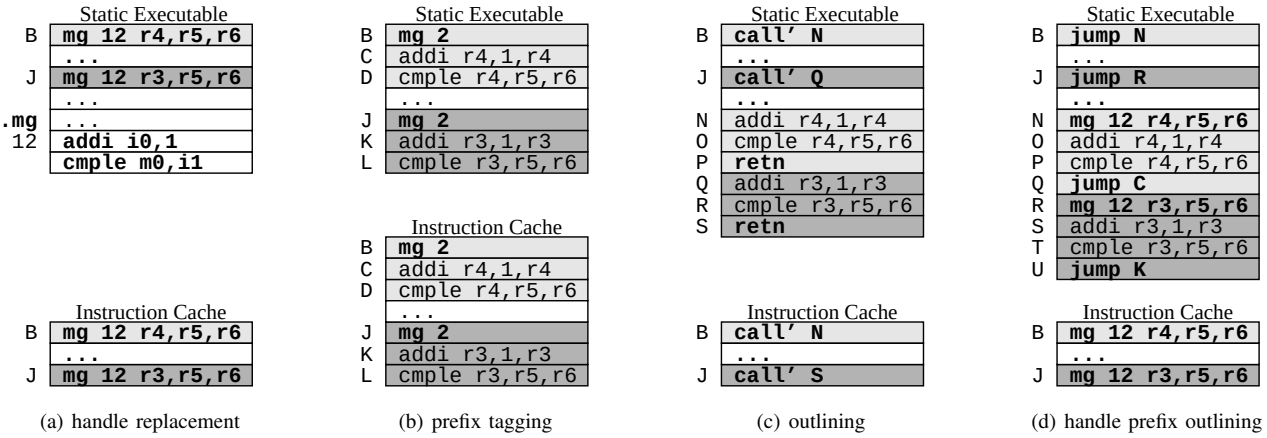
The encoding scheme used by static strands [**?**] leaves mini-graph instruction sequences in their original positions in the code. It prepends each mini-graph with a tag instruction that encodes its length. A mini-graph processor recognizes the tag, accumulates the following instructions into a buffer, and performs the handle-template conversion on the fly. This scheme, which we call prefix tagging, is shown in Figure **??**.

Prefix tagging is trivially compatible across both mini-graph and non-mini-graph processors. And because the original instruction sequences are in the program mainline—both in the binary and the instruction cache—the penalties to fill the MGT and to execute mini-graphs in singleton form are low.

The drawback of prefix tagging is that instruction cache and fetch bandwidth amplification are forfeited. In fact, both are effectively reduced by the presence of tagging instructions. Static Strands overcame this problem by "over-fetching"—fetching two instructions per cycle for an otherwise scalar pipeline. The particular instantiation of prefix tagging used in static strands also doesn't support easy conversion into handle-template format, template sharing, or control over template placement in the MGT. In fairness, the last two criteria are not applicable to the static strands design itself which uses a FIFO dynamic template buffer—analogous to a reorder buffer—as opposed to a cache in which templates persist for multiple invocations.

C. CCA Subgraphs: Outlining

CCA subgraphs are encoded using outlining—as opposed to inlining—or code factoring [**?**], a technique initially proposed for code compression [**?**], [**?**]. In outlining, the mini-graph instruction sequence is replaced in the binary not with the handle but with a special call instruction. The call points to the new location of



(a) handle replacement (b) prefix tagging (c) outlining (d) handle prefix outlining

Fig. 2. Mini-Graph Encoding Schemes. Binary and insruction cache representations.

the outlined instruction sequence, which itself is appended with a return. Outlining is shown in Figure **??**. A mini-graph processor recognizes the special call and fetches the target block from the L2. However, it doesn't place it into the instruction cache. Instead, it scans the block, compiles the instructions (up to the return) into template form, and places them in the MGT. It marks the call's BTB entry with a bit and replaces the call target with the MGT index. On subsequent encounters of the call, the control transfer is suppressed and mini-graph header information from the MGT—the handle—is inserted into the pipeline instead.

Outlining meets both primary criteria. It provides amplification and supports singleton execution of mini-graphs on mini-graph enabled processors. However, it falls short in meeting some of the secondary criteria.

Outlining does not support functional compatibility on non-mini-graph processors because the special outlining call instruction cannot be correctly interpreted as a nop. Its use of call and return supports template sharing but requires mini-graphs that share templates to use the same logical register names. Although a mini-graph aware compiler could orchestrate this, doing so may require additional instructions, defeating amplification. Outlining does not support explicit management of template to MGT set assignment. It also does not help in constructing the handle-template representation because it doesn't explicitly identify the "liveout" register. CCA-subgraphs' authors suggest identifying this register indirectly by adding instructions that explicitly kill the transient registers. Outlining was presented in the context of embedded processors for which both functional compatibility with non-embedded implementations as well as performance compatibility across different embedded implementations is less important. This de-emphasizes features that support explicit MGT management.

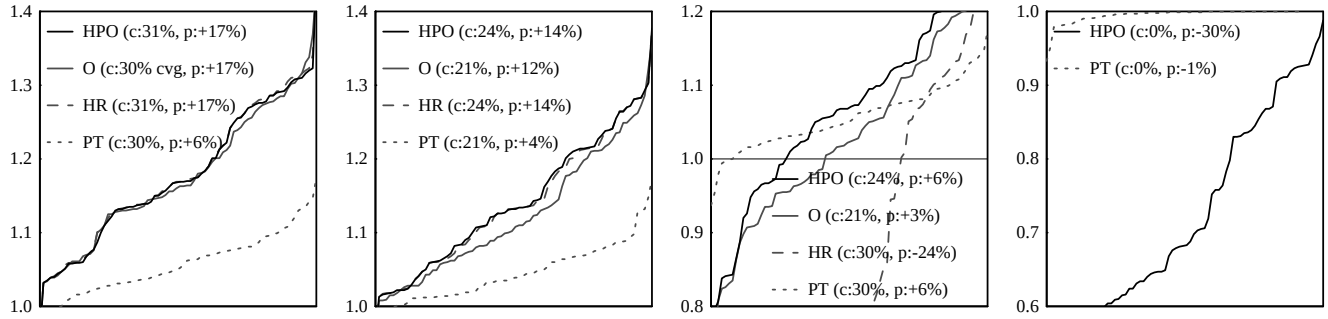
Outlining has moderate singleton execution and MGT miss penalties. In singleton form, each dynamic mini-graph requires two additional instructions, both of which are control transfers. And although both call and return targets are predictable, the resulting fetch discontiguities reduce fetch throughput. The cost of an MGT miss—beyond the cost of template "compilation"—is a single L2 cache access.

D. Mini-Graphs: Handle Prefix Outlining

The encoding scheme mini-graphs use is a hybrid of these three schemes. We give it a fittingly hybrid name—*handle prefix outlining*. Handle prefix outlining is shown in Figure **??**. The basic design is that of outlining, the mini-graph instruction sequence is outlined from the mainline code using a pair of control transfers. However, rather than using a special call instruction and a return, the two instructions are conventional jumps. The fact that the outlined instruction sequence is a mini-graph is denoted by prepending the sequence with a tag instruction—a feature borrowed from static strands. Finally, borrowing from PRISC, the tag instruction is the handle itself, explicitly specifying the interface registers and the template identifier. When the outlining jump is first encountered and resolved and the line containing the mini-graph is fetched from the L2, a mini-graph processor recognizes the handle. It uses the register identifiers in the handle to scan and compile the instruction sequence up to the return jump. The compiled instruction sequence is placed into the MGT at the index indicated by the MGID. Finally, the original outlining jump is over-written with the handle itself. Subsequent fetches of the line encounter the handle instead of the jump. The handle persists in the instruction cache until the containing line is evicted.

Like the classic outlining scheme it derives from, handle prefix outlining satisfies the primary encoding criteria. However, it also satisfies many of the secondary criteria.

Handle prefix outlining is compatible with non mini-graph processors because the instruction that tags the mini-graph is not a control transfer and can be safely interpreted as a nop. It also supports template sharing without the potentially onerous requirment that different static instances of the same template use the same logical register names for both interface and interior registers. In classic outlining, two mini-graphs share a template by "calling" the same outlined instruction sequence. In handle prefix outlining, two mini-graphs share a template if the MGID field in the handle instruction has the same value. In other words, mini-graphs can share templates without sharing static code sequences. This is the reason handle prefix outlining uses conventional jumps rather than a call-return pair. Using a separate outlined static code sequence for each static mini-graphs enlarges the binary, but doesn't increase instruction cache footprint because outlined sequences bypass the instruction cache en route to the MGT. In addition to easy template sharing, the explicit presence



(a) 256 templates, 256-entry MGT (b) 32 templates, 32-entry MGT (c) 256 templates, 32-entry MGT (d) 32 templates, no MGT

Fig. 3. Mini-Graph Encoding Scheme Speedup over Non-Mini-Graph Processor (c = coverage, p = performance).

of the handle in the representation enables control over MGID assignment and eliminates the need for additional instructions to indirectly identify the liveout register.

The downside of handle prefix outlining are that its singleton mini-graph execution and MGT miss penalties are higher than those of classic outlining. Singleton execution requires fetching and decoding the handle in addition to the two jumps. Also, the instruction cache contains handles, not calls, and so on an MGT miss the location of the mini-graph instructions in the binary must itself be fetched from the L2. The cost of handling an MGT miss is therefore two L2 accesses taken in series.

E. Performance Comparison

We use cycle-level simulation to compare the instructions-per-cycle (IPC) performance of these encoding schemes relative to a baseline 3-way fetch/issue/commit non-mini-graph processor. We use the 78 programs from the SPECint2000, MediaBench1.0, CommBench, and MiBench suites. Each encoding experiment is shown using an S-curve; it contains data from all programs sorted worst-to-best. Each S-curve is sorted independently.

The experiment in Figure **??** uses binaries with 256 encoded mini-graph templates and a 256-entry MGT. It isolates bandwidth and instruction cache capacity amplification while minimizing the effects of template sharing and MGT misses. Unsurprisingly, all encoding schemes provide similar coverage (30–31%). Also, handle replacement (HR), outlining (O) and handle prefix outlining (HPO) show the same performance—an average 17% improvement. Prefix tagging (PT), which does not provide fetch bandwidth amplification, has a much lower average performance gain, 6%.

The experiment in Figure **??** uses binaries with 32 encoded templates and a 32-entry MGT. This experiment focuses on amplification but also emphasizes template sharing. Handle replacement and handle prefix outlining support full template sharing. With 32 entries, they achieve coverage rates of 24% and performance improvements of 14%. For outlining, mini-graphs can share templates only if they share logical register names. Our experiments discount the possibility that a mini-graph aware compiler would assign registers to facilitate more template sharing. However, we also do not model additional instructions that would be needed to kill the liveout register. With a smaller MGT, outlining provides 21% coverage and 12% performance.

For the third experiment, shown in Figure **??**, the binary encodes 256 templates but the MGT has space for only 32. Here, both outlining schemes assign MGIDs to templates in order of decreasing coverage and deal with MGT overflow by executing mini-graphs with MGIDs larger than the largest MGT index in singleton outlined form. We give outlining this additional benefit. Despite this additional benefit and the fact that its singleton execution penalty is lower, outlining under-performs handle prefix outlining by 3% on average. Handle prefix outlining's better support for template sharing and superior coverage give it the advantage. The handle replacement scheme does not support singleton execution of mini-graphs—it deals with MGT overflow simply by filling the misses. The cost of this compatibility strategy is severe, however. It results in a 34% average slowdown despite 30% coverage. The performance of prefix tagging is essentially independent of the size of the MGT.

The final experiment, shown in Figure **??**, simulates a binary with 32 encoded templates executing on a non-mini-graph processor. This graph has only two S-cures, because handle replacement and outlining are not compatible with non-mini-graph processors. Handle prefix encoding greatly under-performs prefix tagging in this scenario. However, as we have argued, this is a somewhat contrived scenario that should not be optimized for at the expense of the first two.