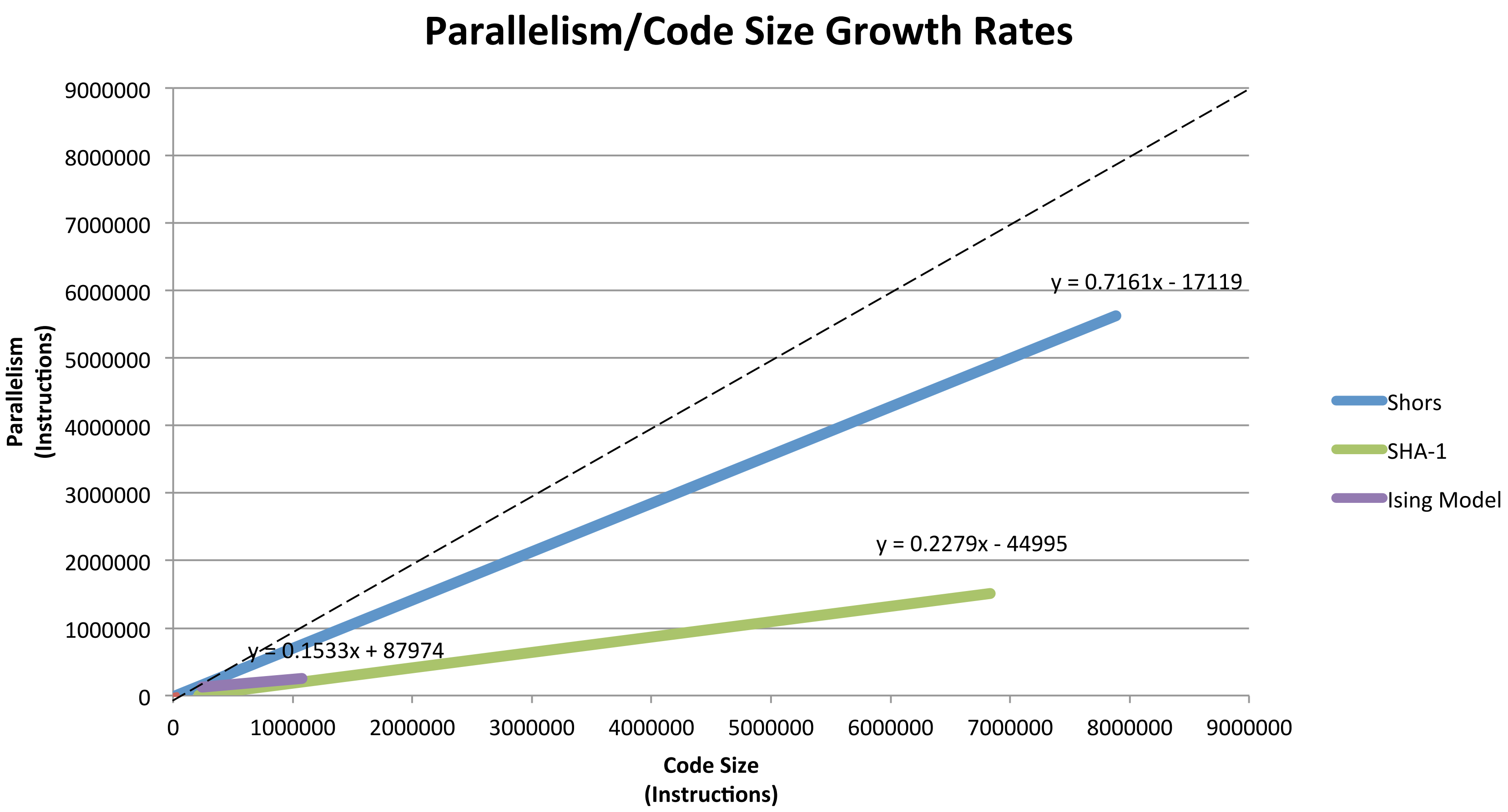
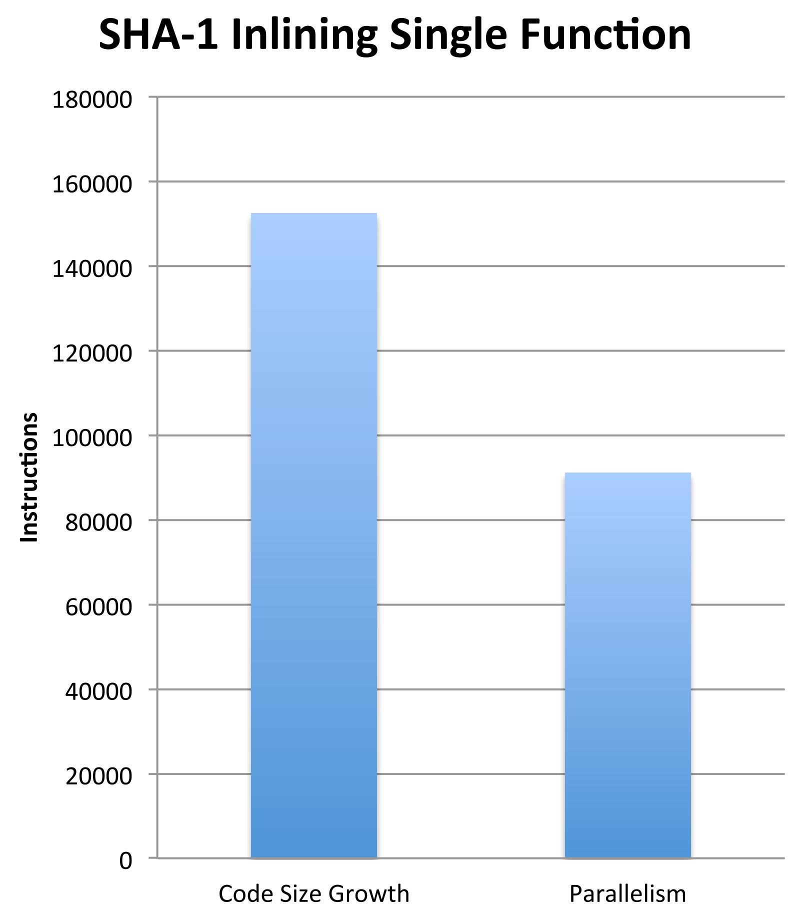
Overview - Motivation



Previous flattening methods lead to over-flattening: parallelism - code size sublinear growth rate

Example: SHA-1

- Consider inlining a single function call
- Details:
  - Most frequently called function
  - Small leaf function
- Fully inlining function leads to over-flattening
  - Nearly 2 lines inlined for each line parallelized
- Trend continues as more functions are inlined

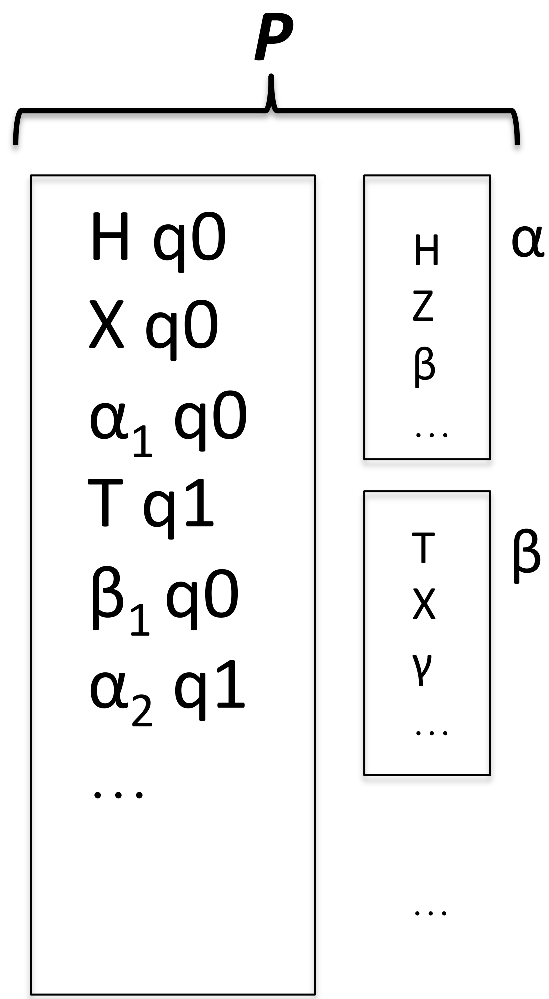


Framework Goals

- Designed to optimize decision to inline a single line from a function call
  - Considers global information about function call
- Can optimally schedule all function call instances of all functions within a program
  - Can be shown that order of applying algorithm on functions does not affect optimality
- Tractable complexity
  - Applying procedure to all functions within an input program, can show $O(\mathbf{n}^3)$ for input program size **n** (fully flattened)
  - Full generality: $2^{|\alpha|}$ subsets for all function call instances $\alpha_i \rightarrow$ exponential complexity
  - By restricting to contiguous increasing-indexed subsequences $\rightarrow O(|\alpha|^2)$ to consider

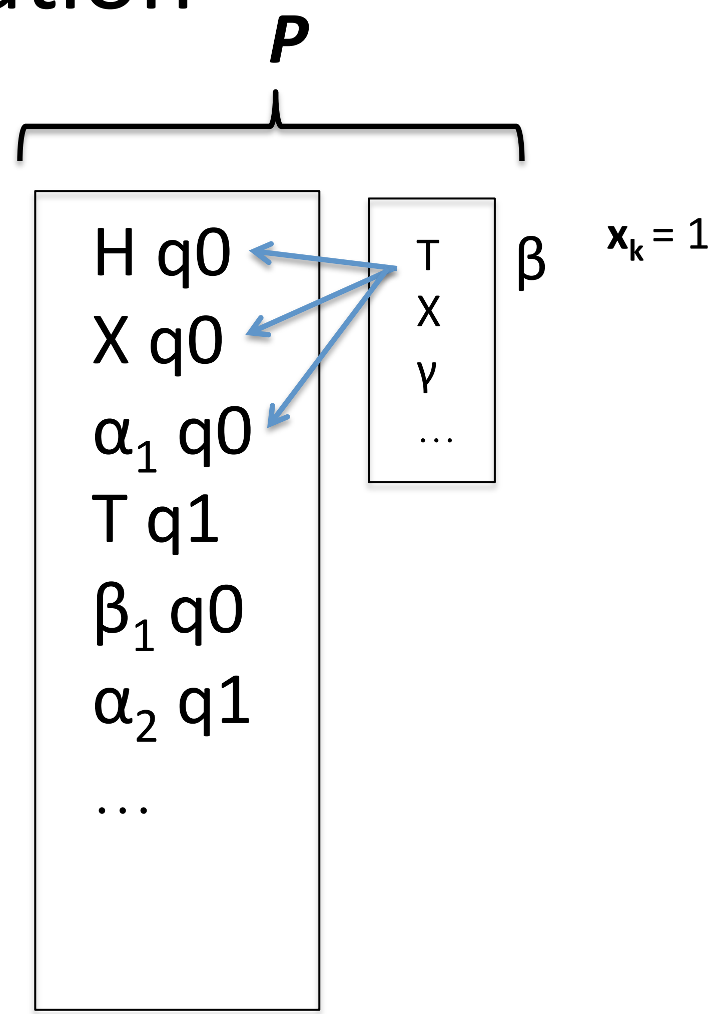
Framework Complexity: Notation

- Input Program: ***P***
  - $|\boldsymbol{P}| = n$
    - all instructions of ***P***
- $\boldsymbol{P_F} = \{\alpha, \beta, \ldots\}$
  - Set of all functions of ***P***
- $\boldsymbol{F} = \{\alpha_1, \beta_1, \alpha_2, \ldots\}$
  - Set of function calls in ***P***
- $\boldsymbol{G} = \{H, X, T, \ldots\}$
  - Set of all intrinsic gates in ***P***

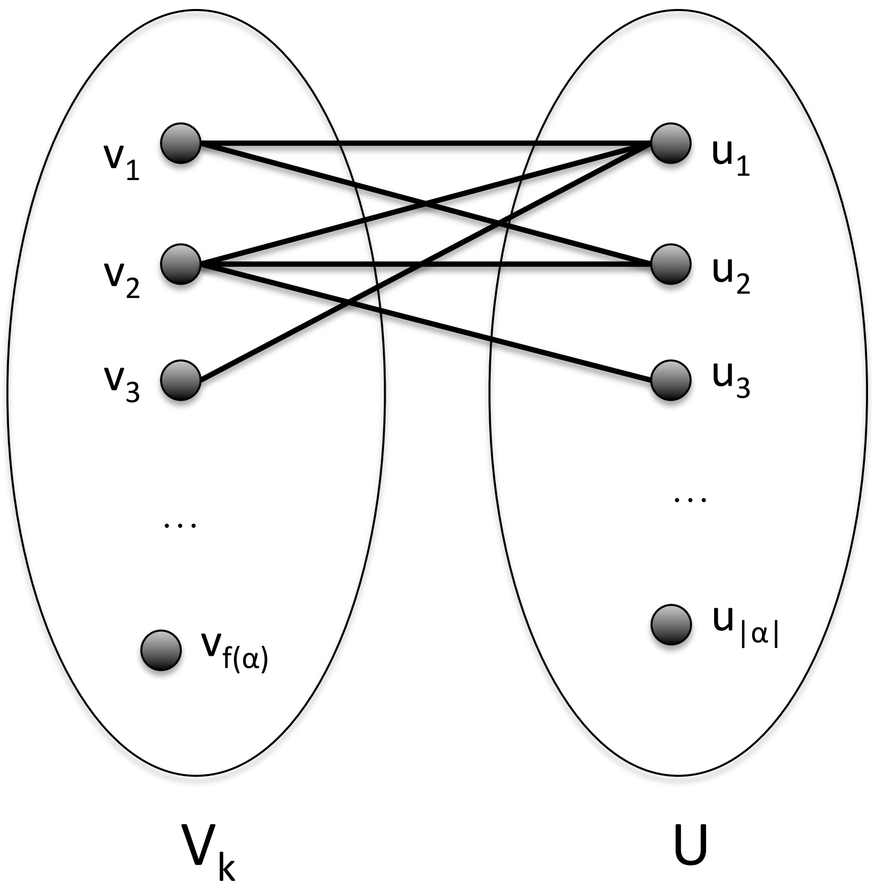


Framework Complexity: Notation

- For each α, choose a starting instruction $\mathbf{x_k}$
- For each instruction **I** in each call site α_i look for compatible instructions **J** in ***P***
  - Can limit this search to insts between latest dependency of **I** and **I** itself
- Partial execution – allow arguments to flow through call sites for accurate dependency analysis



Framework Complexity: Graph



For a given starting instruction $\mathbf{x_k}$ in $\boldsymbol{\alpha}$:
For each call site $\boldsymbol{\alpha_i}$**:**
    For each inst **I** in $\boldsymbol{\alpha}$**:**
        For each inst **J** in $\boldsymbol{P}[\boldsymbol{P_0},\boldsymbol{P_I}]$**:**
            If I $_{||}$ J, return 1

Computing one subgraph $\mathbf{G_k}$**:**
$O(f_\alpha \cdot |\alpha| \cdot n)$

Computing full graph $\mathbf{G_\alpha}$:
$O(f_\alpha \cdot |\alpha|^2 \cdot n)$

Maximum number of edges:
$|E| \leq f_\alpha \cdot |\alpha|^2$

Linear Program Complexity:
$O(f_\alpha \cdot |\alpha|^2)$

Framework Complexity: Full Procedure

- Full Procedure = Build graph + ILP:

$$O(f_\alpha \cdot |\alpha|^2 \cdot n) + O(f_\alpha \cdot |\alpha|^2) = O(f_\alpha \cdot |\alpha|^2 \cdot n)$$

- Over all functions:

$$O(f_\alpha \cdot |\alpha|^2 \cdot n) \text{ for all } \alpha = O(|\boldsymbol{P}|^3)$$

- Overall complexity of running the procedure on all functions in program ***P***:

$$O(|\boldsymbol{P}|^3) = O(n^3)$$

- Loose bound, assumes all instructions must search from beginning of ***P***

Related Work: Complexity

- "Computational complexity of ILP solving grows exponentially with the problem size…we have developed *region scheduling* that allows us to tackle routines of arbitrary size (yet the results are only optimal per region)"
  - *Sebastian Winkel. 2007. Optimal versus Heuristic Global Code Scheduling. MICRO 40*
  - Complexity driven by considering code motion w.r.t loops, branches, and block speculation, and compensation code
  - Principle of Deferred Measurement: our quantum applications can (and are) optimized to defer intermediate conditional measurement-based branching to the end of functions
  - Avoids much of the complexity considered in other work

Objective Function

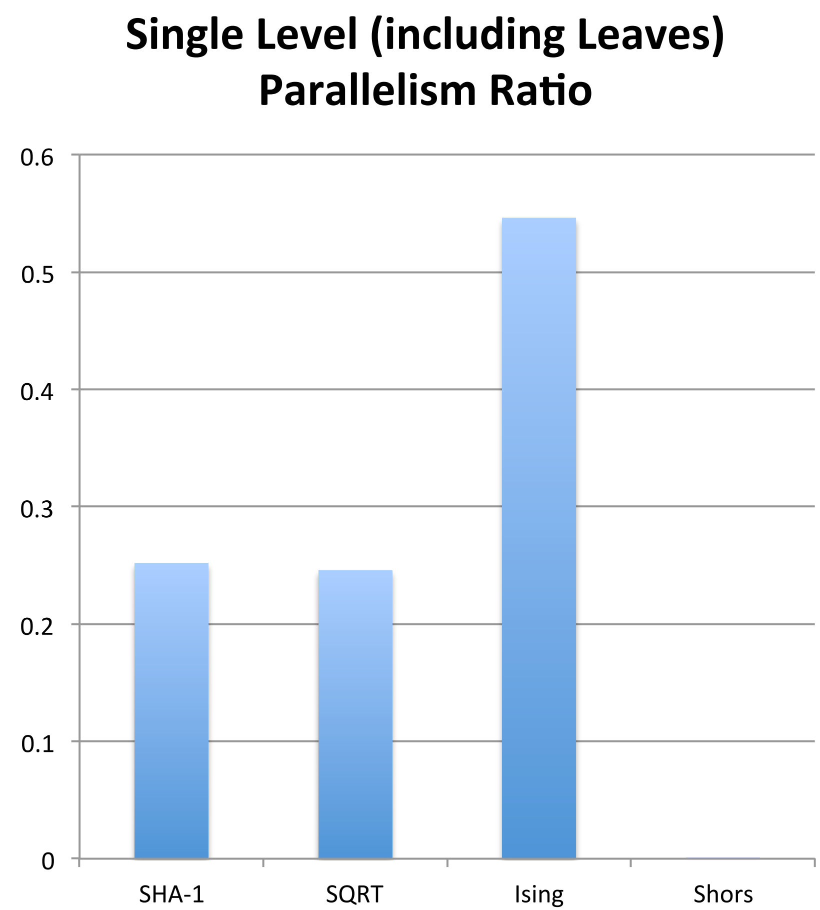
- The objective function on graph **G**: (i_u index of vertex **u**):

$$\max_{X_E \in \{0,1\}^{|E|}} \sum_{j \in [1, f_\alpha]} \sum_{u \in U} \left(\alpha_p \Big(\sum_{e \in E(V_j, U)} X_e \Big) \cdot i_u - \alpha_c i_u \right)$$

- α_P term:
  - Parallelism, number of lines parallelized
- α_c term:
  - Code size penalty to parallelism, size of storing function call
- $\alpha_{P,}$ α_c in [-1,1]: weights allow for tuneability
- Examples:
  - $\alpha_P = 1$, $\alpha_c = 0$: selects $\mathbf{X_E}$ for full parallelism
  - $\alpha_P = -1$, $\alpha_c = 1$: selects $\mathbf{X_E} = 0$ for no parallelism

Framework Weaknesses and Next Steps

- Currently only supports single level hierarchies
  - Avoids recursively applying algorithm on nested function calls
  - Will see most benefit if applied to top level functions and leaves
  - Full generality could be complex to incorporate, but is necessary as only ~25% of parallelism on average is contained in highest level
- Module Reconstruction also suffers from hierarchy weakness



Flattening leaves and top level functions results in 26% of parallelism on average

Reduction: Knapsack to Partial Inlining

- Let $\mathbf{I}_i(\alpha_j)$ denote the inlining choice i for function call α_j
  - $p(I_i(\alpha_j))$ = parallelism gained from this choice
  - $c(I_i(\alpha_j))$ = code size increase from this choice
- Let ***A* = {**set of sets of all inlining choices for function calls α_i **}**
- Given maximum program size **k**, is there a choice of one element from each set contained in ***A*** that maximizes parallelism while total code size bounded by **k**?
- Polynomial in **n** = $\prod_{\alpha \in P} |\alpha|^2$