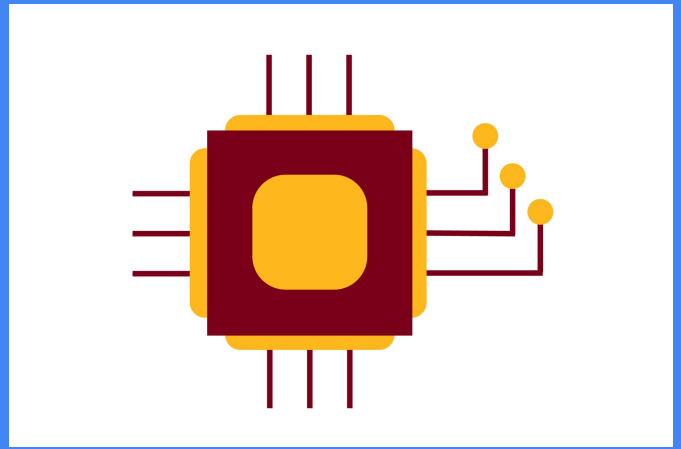
Instrumentación de máquinas Cross-ISA mediante traducción binaria dinámica rápida y escalable

Autor : Juncal Blanco, Aitor



NOVIEMBRE, 2023

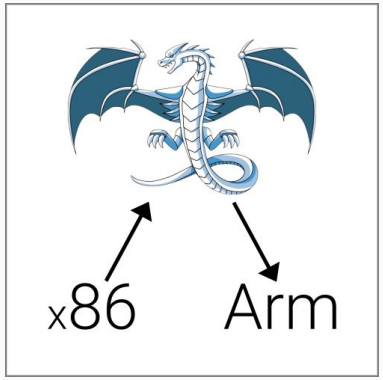
Índice

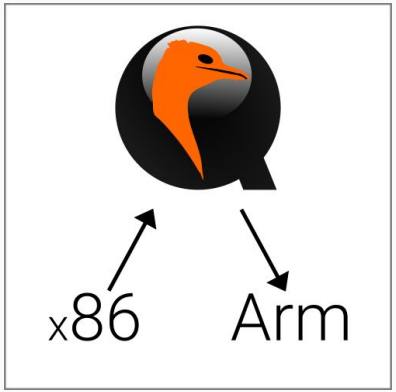
Introduccion

- Emulación cross-ISA : Capacidad de ejecutar código diseñado en una ISA(target) en otra ISA(host).
- DBT
  - Nos permite traducir el código para adaptarlo a la nueva ISA.
  - La compilación de código no es previa a la ejecución, sino que se realiza durante la ejecución
- Instrumentación : Análisis de un programa en tiempo de ejecución para observar su comportamiento.
  - Permite incluso modificar el comportamiento del programa en tiempo de ejecución, agregando nuevas funcionalidades por ejemplo.

Motivación

- Cada vez hay un número mayor de ISAs(x86, ARM, MIPS o RISC-V)
- Creciente uso de máquinas virtuales(emulación de sistema completo)
- Mejorar el rendimiento de la emulación completa cross-ISA
- Mejorar la instrumentación
- Escalabilidad de DBT





Problemas

- Emulación de instrucciones FP
- Escalabilidad de DBT
- Overhead importante introducido por la instrumentación DBI

Problema 1 - Emulación de FP

- Cada arquitectura puede tener una FPU(Floating-point unit)
- Puede ser bastante complejo emular otras FPUs(diferencias en la precisión, en el tratamiento de números denormales o NaN, redondeo, flags…etc)
- La solución que se suele utilizar es la emulación soft-float
- Se emula una FPU a partir de operaciones en números enteros(ALU)
- La correcta emulación de una FPU de otro ISA trae consigo mucho overhead

Problema 2 - Escalabilidad DBT

- La escalabilidad multicore no es uno de los principales objetivos de DBT
- Se puede sufrir mucho en aplicaciones que requieren alto paralelismo y en grandes servidores multicores.
- Se suelen utilizar caches de código compartido
  - Múltiples accesos concurrentes
  - Locks para garantizar la coherencia de la cache
- Las caches de código privada por hilo (subproceso) tampoco solucionan el problema
  - Se consigue mejorar la escalabilidad
  - Pero el uso de memoria puede ser prohibitivo

Problema 3 - Overhead introducido por DBI

- La instrumentación DBI implica insertar/analizar código adicional (instrumentación) en el programa ejecutado dinámicamente
- La instrumentación requiere de nuevas instrucciones
- Aumenta el ya grande uso de memoria
- Aumento en el tiempo de ejecución
- Problemas no deseados(pero inevitables) introducidos por la instrumentación
  - Queremos medir secciones críticas del código(accesos a memoria, tiempo..etc)
  - Necesitamos un código de instrumentación para estas medidas.
  - Ese código puede aumentar en mayor o menor medida lo que queremos medir.

Soluciones

- Se presenta Qelt como herramienta
- Emulador cross-ISA de alto rendimiento
- Permite instrumentación
- Desarrollada por los autores de este paper
- Basada en Qemu y mejorada para los objetivos de este paper

Solución 1 - Emulación de FP

- La idea es aprovechar al máximo la utilización de la FPU del host, sabiendo que la mayoría de operaciones de FP dan el mismo resultado en la FPU del host y la FPU del guest.
- Problema con el manejo de flags de la FPU => Induce pérdidas de rendimiento
- Casos excepcionales problemáticos
  - Números denormales
  - NaN
  - Infinitos
- ¿Qué modo de redondeo hay que usar?

Se busca mejorar el caso común

Los NaN o denormales son casos muy poco frecuentes, nos centramos en los casos más frecuentes(normales y ceros)

Tratamos de emular las excepciones más frecuentes(inexact, underflow, overflow)

La flag de inexactitud es la más común

Las flags FP pueden ser acumulativas y son borradas explícitamente por software

El modo de redondeo más común es round-to-nearest-even

Si contemplamos un caso fuera de lo normal, recurrimos a la implementación soft float

Ejemplo

```
float64 float64_mul(float64 a, float64 b, fp_status *st)
{
  float64_input_flush2(&a, &b, st);
  if (likely(float64_is_zero_or_normal(a) &&
             float64_is_zero_or_normal(b) &&
             st->exception_flags & FP_INEXACT &&
             st->round_mode == FP_ROUND_NEAREST_EVEN)) {
    if (float64_is_zero(a) || float64_is_zero(b)) {
      bool neg = float64_is_neg(a) ^ float64_is_neg(b);
      return float64_set_sign(float64_zero, neg);
    } else {
      double ha = float64_to_double(a);
      double hb = float64_to_double(b);
      double hr = ha * hb;
      if (unlikely(isinf(hr))) {
        st->float_exception_flags |= float_flag_overflow;
      } else if (unlikely(fabs(hr) <= DBL_MIN)) {
        goto soft_fp;
      }
      return double_to_float64(hr);
    }
  }
soft_fp:
  return soft_float64_mul(a, b, st);
}
```

Conclusiones de la solución

- El enfoque aprovecha la FPU del anfitrión para un conjunto de operaciones de punto flotante, cubriendo la mayoría de las instrucciones de punto flotante en el código del mundo real.
- La técnica identifica un gran conjunto de operaciones de punto flotante adecuadas para la FPU del anfitrión, posponiendo casos específicos a un código de soft-float más lento.
- Qelt implementa esta técnica, acelerando operaciones comúnmente utilizadas de precisión simple y doble: suma, resta, multiplicación, división, raíz cuadrada, comparación y multiplicación-adición fusionada.
- Qelt acelera el 99.18% de las instrucciones de punto flotante en las workloads de SPECfp06

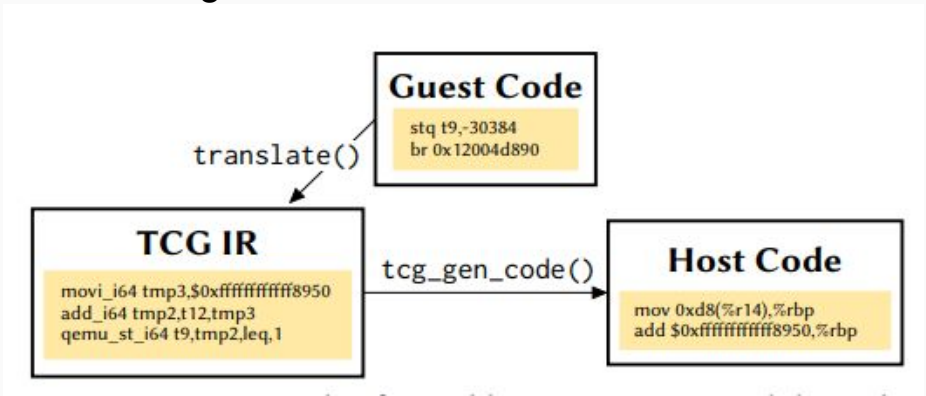
Solución 2 - Escalabilidad con DBT

Los motores DBT de última generación si permiten la ejecución paralela de huéspedes con una caché de código compartida.

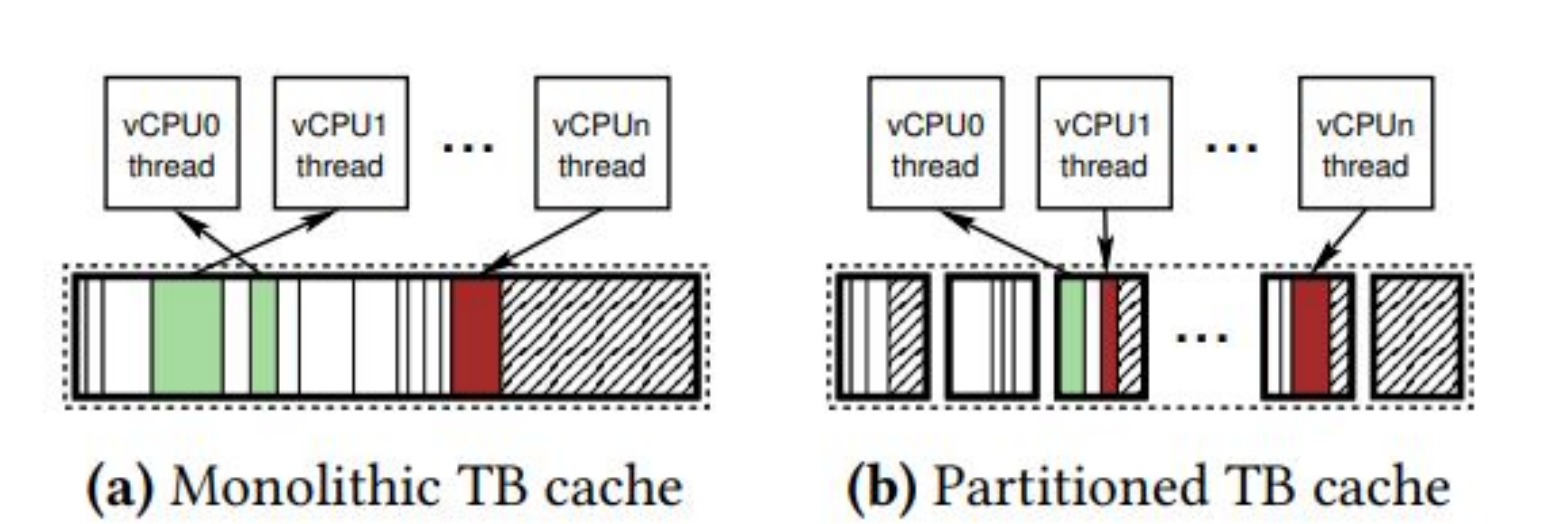
Enfocarse en la ejecución paralela de código es beneficioso, ya que la mayoría del tiempo de ejecución en cargas de trabajo DBT se gasta en la ejecución de código.

Lograr la escalabilidad para traductores de caché de código unificada es desafiante debido a la contención impuesta por locks.

Dividimos la solución en 3 partes



Estado del traductor y caché de código

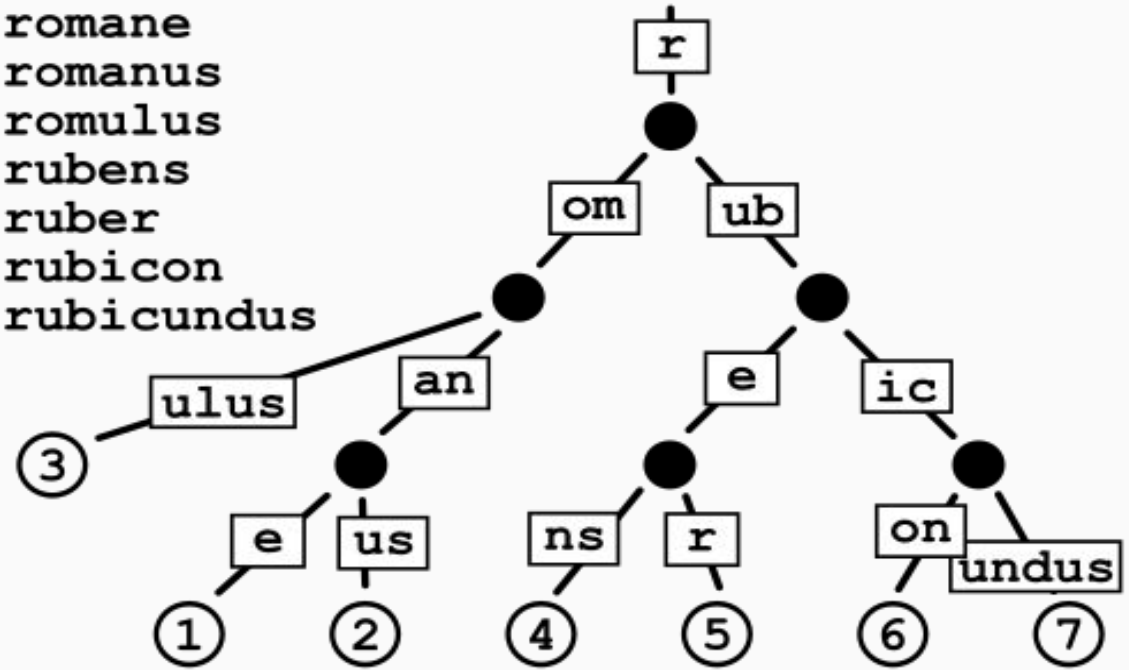


Mapeo de memoria física

- Se utiliza paginación para la gestión de memoria virtual
- Radix tree: Árbol que organiza la información de manera jerárquica
- Inserciones Lock-free: Para mejorar la eficiencia y evitar bloqueos innecesarios, se modificó el árbol radix para admitir inserciones sin necesidad de bloqueo.
  - RCU (Read-Copy-Update): Se utiliza RCU para consultas en el árbol. RCU es una técnica que permite realizar lecturas concurrentes sin bloqueo mientras se actualiza la estructura de datos.
- Spin lock: Se agrega un spin lock a cada descriptor de página para serializar el acceso a la lista de TBs con el fin de que las operaciones de añadir o quitar TBs sean seguras en entornos concurrentes.
- Cada descriptor de página tiene una lista de TBs asociados. Durante la traducción de código o la invalidación, se accede a esta lista cuando se añaden o eliminan TBs.

RADIX TREE

1 romane
2 romanus
3 romulus
4 rubens
5 ruber
6 rubicon
7 rubicundus



Encadenamiento de código

Buscamos vincular TBs que se relacionan mediante saltos

Parcheamos el código generado para realizar los saltos entre TBs directamente

Cada salto está protegido con spin-lock

TBs etiquetados para evitar saltos inválidos/prohibidos

Solución 3 - Instrumentación

- Objetivo => Convertir un motor DBT cross-ISA en una herramienta cross-ISA
- Puntos de Inyección a Nivel de Instrucción: la instrumentación puede ocurrir en puntos específicos dentro de las instrucciones
- ISA-Agnostic: No se limita a una única ISA
- Eventos: dinámicos y regulares(Manejados mediante plugins)
  - Los eventos regulares son aquellos que no están relacionados con la ejecución de la máquina virtual, como el inicio o la finalización de un hilo de vCPU o la traducción de bloques de traducción (TB).
  - los eventos dinámicos están relacionados con la ejecución de la máquina virtual, como el acceso a la memoria o la ejecución de bloques de traducción.

Solución a los eventos dinámicos

- Inyección de código de instrumentación vacío
  - durante la traducción de bloques de traducción (TB), se inyecta código de instrumentación vacío en el TB.
  - sirve como marcador para indicar que se debe llamar a una devolución de llamada de instrumentación en un punto específico del TB
  - Mejora el rendimiento de los eventos dinámicos al reducir la cantidad de devoluciones de llamada que se deben realizar. En lugar de llamar a una devolución de llamada para cada evento dinámico, se llama a la devolución de llamada solo cuando se alcanza el marcador en el TB.
- Devoluciones de llamada directas:
  - En lugar de utilizar una devolución de llamada indirecta, que implica llamar a una función de ayuda que luego llama a la devolución de llamada final, las devoluciones de llamada directa llaman directamente a la devolución de llamada final desde el punto de instrumentación.

¿Pero cómo realizamos la instrumentación? Plugins

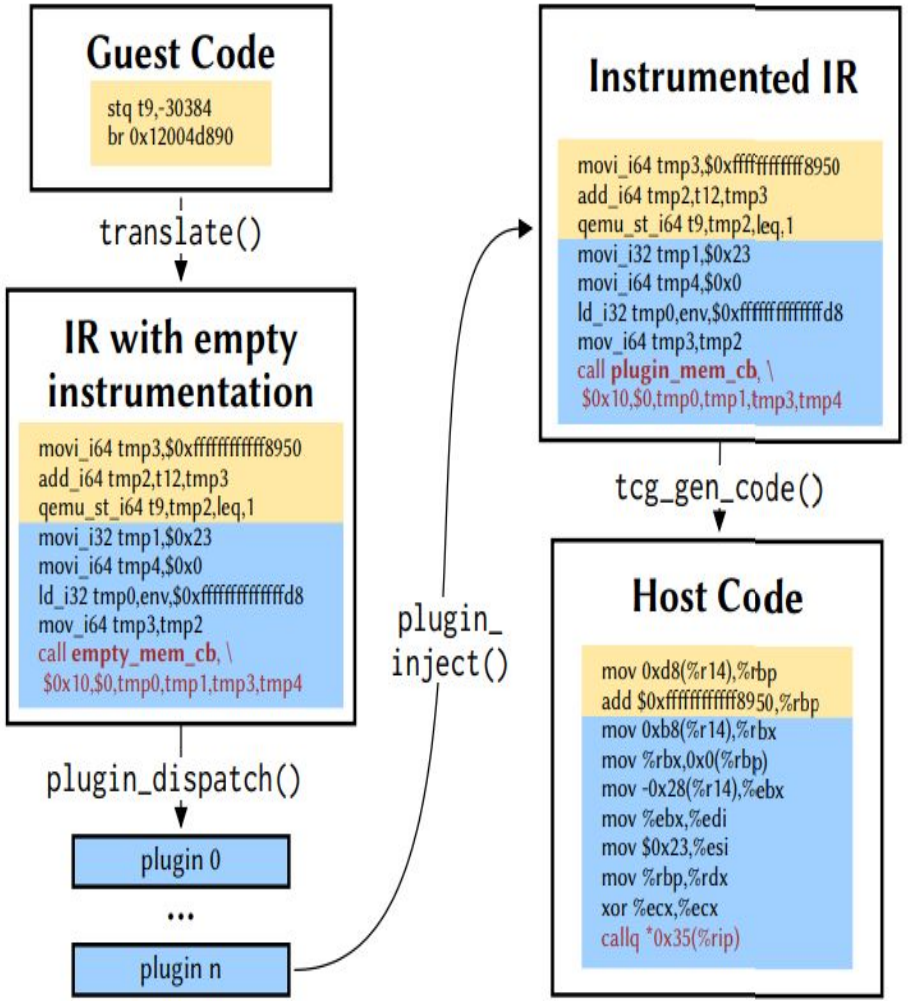
Son módulos de software que se utilizan para agregar funcionalidad de instrumentación

Los plugins de instrumentación se registran en la máquina virtual y se suscriben a eventos específicos utilizando devoluciones de llamada.

Cuando se produce un evento, la máquina virtual llama a las devoluciones de llamada correspondientes.

Permite que los plugins de instrumentación realicen alguna acción en respuesta al evento.

Otro método => Inlining => Inserción de código de instrumentación directamente en el código de la máquina virtual en lugar de llamar a una función de instrumentación separada

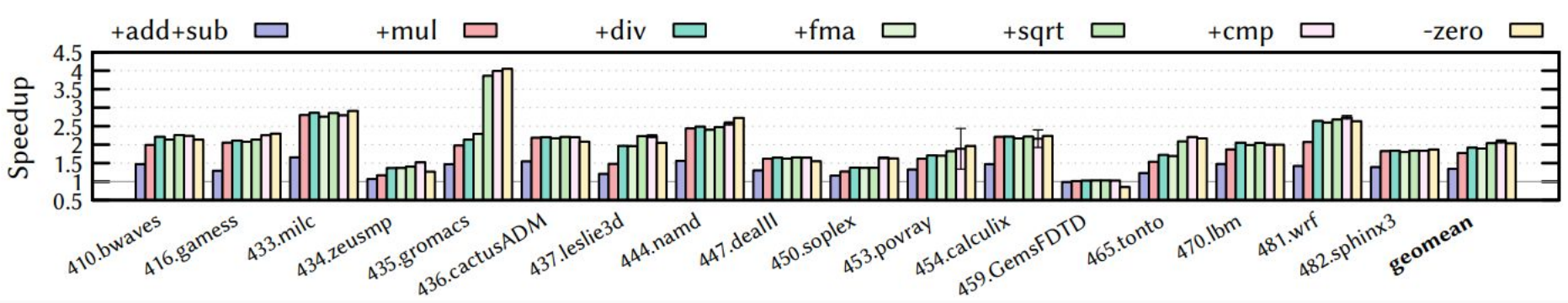


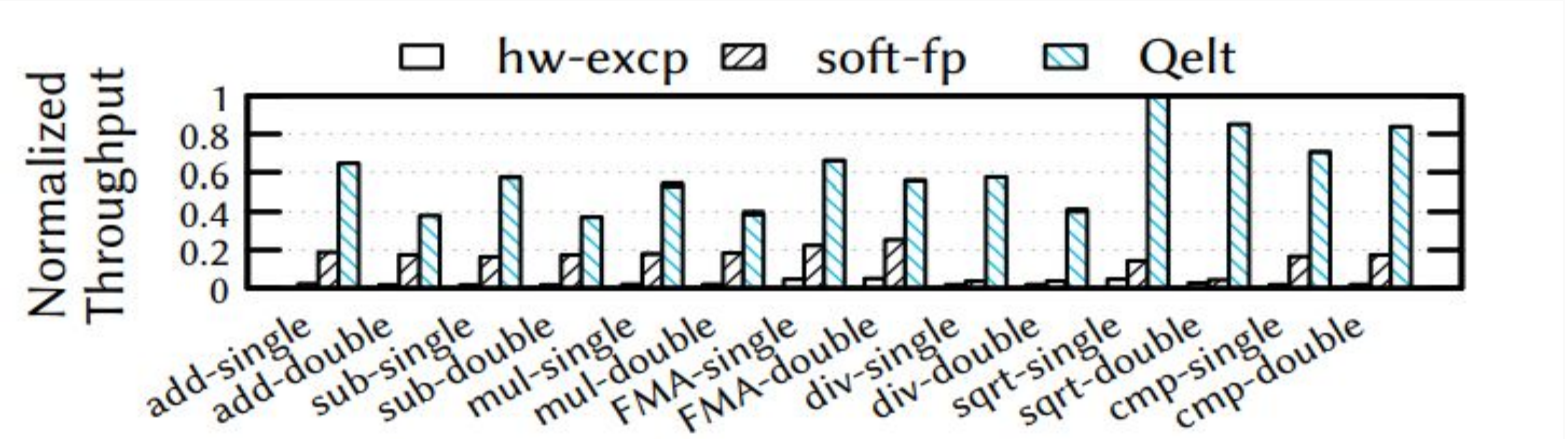

```c
static uint64_t mem_count;
static bool do_inline;

static void plugin_exit(plugin_id_t id, void *p)
{
  printf("mem accesses: %" PRIu64 "\n", mem_count);
}

static void vcpu_mem(unsigned int cpu_index,
  plugin_meminfo_t meminfo, uint64_t vaddr, void *udata)
{
  mem_count++;
}

static void vcpu_tb_translate(plugin_id_t id,
  unsigned int cpu_index, struct plugin_tb *tb)
{
  size_t n = plugin_tb_n_insns(tb);
  size_t i;
  for (i = 0; i < n; i++) {
    struct plugin_insn *insn = plugin_tb_get_insn(tb, i);
    if (do_inline) {
      plugin_register_vcpu_mem_inline__after(
        insn, PLUGIN_INLINE_ADD_U64, &mem_count, 1);
    } else {
      plugin_register_vcpu_mem_cb__after(
        insn, vcpu_mem, PLUGIN_CB_NO_REGS, NULL);
    }
  }
}

int plugin_install(plugin_id_t id, int argc, char **argv)
{
  if (argc && strcmp(argv[0], "inline") == 0)
    do_inline = true;
  plugin_register_vcpu_tb_trans_cb(id, vcpu_tb_translate);
  plugin_register_atexit_cb(id, plugin_exit, NULL);
  return 0;
}
```

Evaluación - Setup

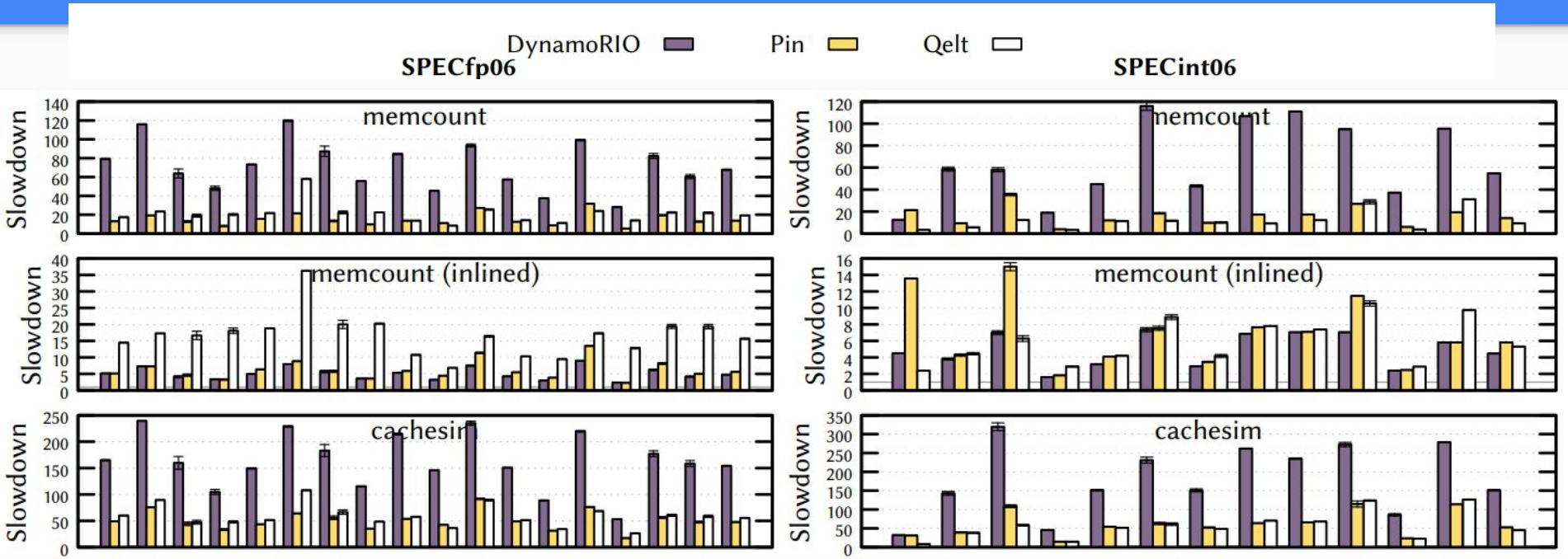
- 2 socket con 16 cores cada uno
- Intel Xeon Gold 6142 processors
- 384GB de ram
- Ubuntu 18.04 | kernel v4.15.0.
- SPEC 06
- Cross validation
- Intervalo de confianza del 95%

Emulación rápida de FP

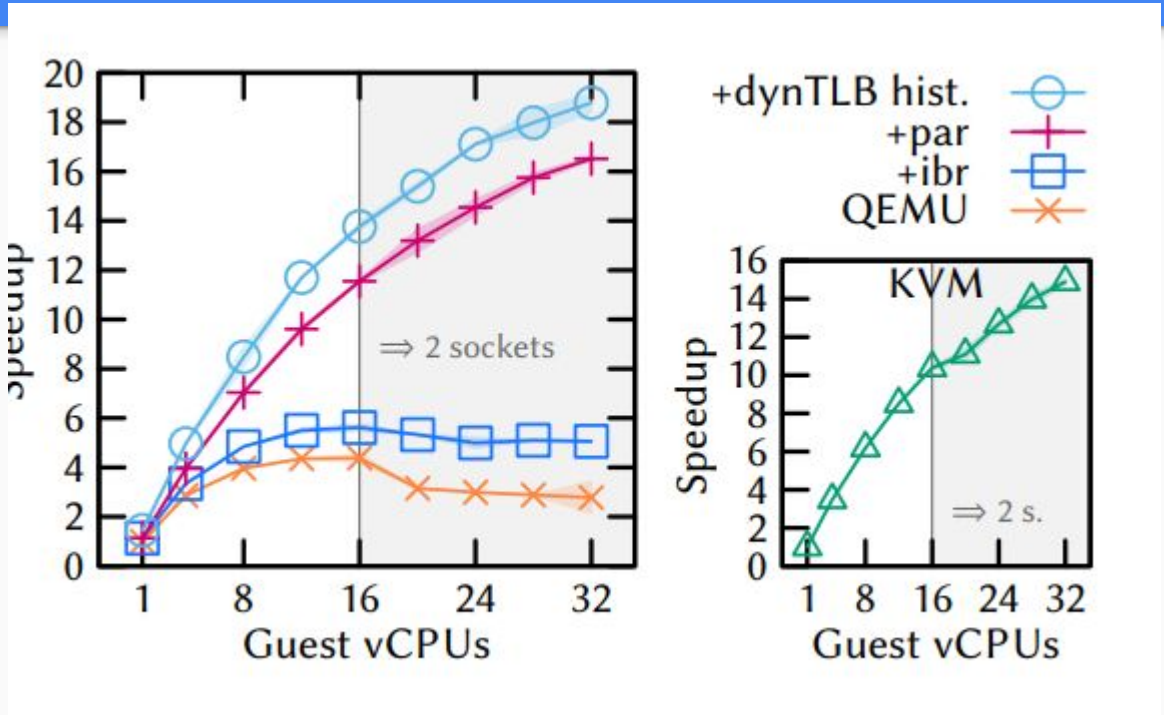




DBI


DynamoRIO
Pin
Qelt
SPECfp06
SPECint06
memcount
memcount (inlined)
cachesim
Slowdown

Escalabilidad



Conclusiones de los autores

Aumento del Rendimiento: emulación rápida de FP

Escalabilidad: traducción DBT escalable

Instrumentación “ISA-agnostic”: permite convertir motores DBT cross-ISA en herramientas DBI cross-ISA.

Qelt : un emulador de máquina cross-ISA y herramienta DBI que supera a los competidores state of art

Conclusiones propias

Paper sencillo de leer

Resultados efectivamente positivos

Se centran en el caso común para las mejoras de Qelt

¿Merece la pena atacar los corner cases?

¿Uso de Qelt en entorno industrial?